

Эта книга станет для Вас чрезвычайно
полезным помощником в работе
по программированию в Windows.
Филипп Кан

П. Нортон
П. Иао

Программирование на BORLAND C++ в среде WINDOWS



Программирование на Borland C++ в среде Windows

П.Нортон, П.Йао

том 1



**Киев
"Диалектика"
1993**

ББК 32.973

Н57

Н57 Программирование на Borland C++ в среде Windows: В 2-х томах: Том 1.-

К.: "Диалектика", 1993. - 320 с., ил.

ISBN 0-553-35143-5 (англ.)

ISBN 5-7707-5385-4 (том 1)

ISBN 5-7707-5384-6

В книге двух всемирно известных специалистов в области персональных компьютеров - Питера Нортон и Поля Яо - описывается техника эффективного использования Borland C++, библиотеки ObjectWindows и функций API Windows (Application Programing Interface). В ней приводятся тексты программ, предназначенные для более полного ознакомления с этими средствами и их возможностями. Однако эта книга не просто набор листингов программ, в ней также подробно рассматриваются концепции и философия Microsoft Windows.

Книга рассчитана на программистов разной квалификации, преподавателей, студентов и всех, кто разрабатывает программы для среды MS Windows с использованием Borland C++ и библиотеки ObjectWindows.

ББК 32.973

Группа подготовки издания:

Редакторы

П.В.Гусак, Ю.Ю.Плаксюк

Литературные редакторы

А.П.Петриковец, А.В.Штин, И.А.Шишкина

Технические редакторы

Т.Л.Юрчук,

Н.Е.Костина

Художник обложки А.С.Сторожук

Рассылка по почте А.Н.Тимченко

Учебное издание

Программирование на Borland C++ в среде Windows

Подписано к печати 22.11.1993. Формат 70x108 1/16. Бумага офсетная. Гарнитура Таймс. Печать офсетная. Усл.печ.листов 28.

Заказ N 0214033. Цена договорная.

Подготовлено к печати издательским отделом фирмы "Диалектика".

Украина, 252124 Киев-124, а/я 506. Телефон/факс (044) 266-4074.

ISBN 0-553-35143-5 (англ.) Copyright © 1992 by Peter Norton and Paul Yao
Cover Design © 1992 by Bantan Books

ISBN 5-7707-5385-4 (том 1) © НИПФ "Диалектика", 1993, обработка и оформление

ISBN 5-7707-5384-6

Borland® является зарегистрированной торговой маркой фирмы Borland Int.
Windows™ является торговой маркой фирмы Microsoft Corporation.

Отпечатано с компьютерного набора на комбинате печати
издательства "Пресса Украины".

Украина 252047, Киев-47, проспект Победы, 50

Предисловие

Одним из главных препятствий при разработке прикладных программ Windows является необходимость обработки большого количества обращений от интерфейса прикладных программ (API) Windows. Хотя мы и почувствовали благодаря Borland, что наша библиотека ObjectWindows в значительной степени упрощает задачу успешного взаимодействия с API, для создания оптимальных и эффективных прикладных программ, полностью соответствующих стандартам Windows, необходимо знать, как Windows обеспечивает такое взаимодействие и как рассматривать отдельные ее аспекты.

Мы чрезвычайно рады приветствовать двух всемирно известных специалистов в области персональных компьютеров - Питера Нортон и Поля Йао - в контексте данной проблемы как официальных авторов Bantam/Borland. Питер стал легендой благодаря своей способности объяснять на народном английском языке все "гаечки и винтики" в вопросах практической работы, тогда как Поль привнес теоретическую глубину в понимание возможностей Windows, и это говорит о том, что, продолжая вести конкурентную борьбу, соперники могут совместными усилиями найти верный подход.

Данная книга рассчитана на программистов, поскольку даже мимолетный взгляд на пространные листинги производит впечатление. Но это не просто совокупность огромной массы листингов - здесь достаточно подробно рассмотрены и концептуальные вопросы Windows. Borland C++ дает Вам инструмент для овладения Windows, Питер и Поль помогают Вам в приобретении навыков, необходимых для эффективного применения этого инструмента. Сконцентрировав все внимание на библиотеке OWL, они показали, каким образом Вы можете быстро и эффективно создать свою собственную прикладную программу, используя полный диапазон возможностей Windows - тексты, графику, меню, блоки диалога, полосы прокрутки и многое другое.

Мы надеемся, что эта книга станет для Вас чрезвычайно полезным помощником в работе по программированию в Windows.

*Филипп Кан
Philippe Kahn,*

Главный управляющий компании Borland International

Вводная часть

Через несколько месяцев после того как мы представили книгу П.Нортон "Windows 3.0 Power Programming Techniques" для языка программирования С, фирма Borland International объявила о создании компилятора Borland C++ для Windows. В то время как большинство разработчиков Windows сегодня используют С, мы предсказываем преобладание С++ в перспективных разработках. Что касается разработчиков С, то для них С++ является средством последовательного плавного перехода к дополнительным средствам объектного программирования для существующих систем, написанных на С. Поэтому мы создали версию С++ нашей книги. Мы выбрали компилятор Borland C++ и библиотеки Borland ObjectWindows (OWL), поскольку считаем, что они представляют собой хорошую основу для построения прикладных программ. Фундаментальные положения программирования Windows остаются прежними, независимо от используемого языка. В связи с этим основная часть книги почти идентична версии для С. Исключения для данного утверждения заключаются в следующем:

- * Главы 2, 3, 4 и 5 были написаны полностью для редакции С++. Это является отражением того факта, что "минимальная" программа Windows, написанная на С, довольно резко отличается от подобной ей программы, написанной на С++ с использованием OWL. Среди прочих тем эти главы содержат описание основной организации двух исходных объектов OWL: объекта прикладной программы и объекта окна.

- * Все примеры программ написаны в С++. Они позволяют понять как программирование на С++ может сочетаться с программированием на Windows.

Главная цель этой книги - помочь опытным программистам, работающим на С++, научиться писать программы в Windows. Существует несколько исходных положений, являющихся бесценными при попытке разобраться в новой среде. Одно из них — образец кодирования, демонстрирующий ключевые моменты технологии, необходимые для понимания преимущественных возможностей среды. Второе заключается в мудром совете от лица наставника, который знает все входы и выходы данной среды. В этой книге есть и то, и другое.

Мы написали фрагменты программ для демонстрации использования различных возможностей Windows и приложили полные листинги этих программ, чтобы дать Вам полное представление о том мире, с которым Вам придется иметь дело. Кто-то сказал, что была написана вообще лишь одна программа Windows. Все остальные программы Windows можно назвать "перелатыными обрезками" самой первой. Не стесняйтесь и Вы использовать любой знакомый Вам код в своих рабочих проектах Windows (эти программы представлены в форме, удобной для машинного чтения. О подробностях Вы сможете узнать, раскрыв внутреннюю сторону обложки).

Для того чтобы начать программировать в Windows, Вам будет необходимо следующее:

- * Один из трех программных продуктов Borland: Turbo C++ для Windows; Borland C++ для Windows либо Borland C++ и Application Frameworks.

- * Windows версии 3.0 или более поздние.

Если Вы планируете работать с Turbo Debugger, то также могут потребоваться следующие аппаратные средства, благодаря которым Turbo Debugger получит собственное устройство вывода:

- * Адаптер MDA и монохромный монитор для компьютера, оснащенного шиной типа AT. Часто может использоваться вместо адаптера MDA графический адаптер, совместимый с Hercules.

- * Адаптер 8514/а и совместимый дисплей (кроме имеющегося монитора типа VGA) для компьютера, оснащенного шиной MCA.

С темой программирования в Windows в книге тесно переплетается описание архитектуры Windows. Создавая первую редакцию книги, мы обращались к помощи

коллектива, разрабатывающего Windows, и не имели представления о внутренней работе Windows. Данная версия книги - это результат сотрудничества с разработчиками OWL, благодаря кому мы разобрались в вопросе разработки классической библиотеки OWL.

Эта книга разделена на шесть частей. В части I мы приводим краткую историю Windows и рассматриваем три коллизии, с которыми пришлось столкнуться новым программистам: программирование с управляющими сообщениями, генерацию и управление графическим выводом и изучение того, каким образом использовать различные объекты интерфейса пользователя.

В части II мы даем анализ "минимальной" программы Windows и раскрываем несколько фундаментальных тем программирования в Windows. Цель данного раздела — изучение основных структур, которые имеют отношение ко всем программам Windows. Вы убедитесь в том, что программа Windows OWL представляет собой сочетание, как минимум, двух компонентов: прикладного объекта и объекта окна.

Часть III познакомит Вас с интерфейсом графических устройств (GDI) Windows. Ваши программы будут использовать GDI для осуществления аппаратно-независимого графического вывода на экраны, принтеры и плоттеры. Описываются базовые концепции программирования графики, в том числе дается представление о координатах прорисовки, контексте устройства GDI и о вырезке изображения (глава 6). Отдельные главы посвящены созданию посредством GDI пикселей, линий, закрашенных поверхностей и текста.

В части IV речь идет о трех ключевых объектах пользовательского интерфейса: меню, окнах и блоках диалога. Мы даем описание внутреннего механизма работы каждого типа объекта и всех основных средств для обеспечения оперативности их работы.

Часть V подробно излагает вопрос пользовательского ввода. Система Windows является операционной средой, где управление осуществляется посредством сообщений, поэтому вполне логично обнаружить, что результат ввода имеет форму сообщений. Далее приводится описание потока данных от реальных аппаратных средств через системные буферы в программу Windows.

В части VI рассматриваются соглашения операционной системы. Сюда включены два основных раздела: память и динамическая компоновка. Один из крупных новых элементов Windows 3 включает в себя возможности совершенствования процесса использования памяти. Для того чтобы Вы смогли иметь точное представление о том, что это означает, мы постарались дать описание внутреннего механизма работы каждого рабочего режима Windows.

БЛАГОДАРНОСТЬ

Поиск и обработка объема информации, необходимой для такой книги, как эта, является задачей, выполнение которой становится нереальным делом без помощи большого числа людей. Мы счастливы быть связанными с целой группой специалистов, разделивших невероятные приключения, приведшие к успеху Windows.

Мы рады поблагодарить в первую очередь группы разработчиков Windows 1.x, 2.x и 3. Особенно нам хочется выразить признательность тем специалистам, которые отдали свое время и силы, чтобы помочь нам понять самые хитрые тонкости работы Windows. Это Питер Белью, Джон Бутлер, Марк Клингет, Кларк Кир, Рик Дилл, Мэрлин Эллер, Рон Гери, Боб Гандерсон, Поль Клиггер, Скотт Макгрегор, Эд Милс, Уолт Мур, Гэйб Ньюел, Крис Питерс, Джон Полок, Рэо Ремала, Лин Шэу, Чарльз Симони, Тэнди Трувер, Мэни Велон, Дэвид Вайс и Стив Вуд.

Далее, выходя за пределы Microsoft, благодарим тех, кто помогал и поддерживал нас при создании этой книги. Во-первых, особая благодарность Давиду Дюранту, который был и остается нашим первейшим инструктором в программировании Windows. По адресу Application's Development Division шлем нашу благодарность Джиму Кэшу - он подсказал, как сделать понятным процесс программирования Windows, а также Полю Клемонду, прикладному разработчику Windows, мастеру

своего дела, который помог нам решить немало загадок в программировании Windows.

Что касается Borland International, то нашу благодарность за их поддержку в этом проекте обязаны принять: Юджин Ванг, Нэн Боресон и Чарльз Дикерсон. Спасибо Глену Кохану за исчерпывающую техническую редакцию этой книги; спасибо Питеру Идену и Эрику Свенсону за разъяснение нам некоторых деталей разработки компилятора Borland C++ и библиотек Borland OWL.

Особые слова благодарности должны принять Ребека Брокард и Джим Трейси из фирмы International System Design, они не отказались вложить частицу своего таланта в этот проект.

Нам бы хотелось высказать большую благодарность нашему издателю и вице-президенту издательства Bantam Electronic Publishing Кензи Цугихара. Благодарим также Стефена Гьюти, нашего редактора, за кропотливую работу. Большое спасибо Лаурелли Бадер, Джифу Рину и Тому Залкиевичу, чьими руками была создана сама эта книга.

И наконец, горячую признательность выражаем Нэнси и Элен из Rogacha Restaurant, официальным "кормилицам" этой книжки.

Часть 1. Введение в Windows

Глава 1. Введение в Windows

Система Microsoft Windows представляет собой графическое расширение операционной системы MS DOS. Windows расширяет DOS в нескольких направлениях. Система DOS поддерживает одновременное выполнение всего одной программы, в то время как Windows позволяет параллельно выполнять несколько программ сразу. Операционная система DOS имеет ограниченные средства поддержки графического вывода, Windows же способна поддерживать сложную, высокоуровневую графику. DOS требует, чтобы каждая программа обеспечивала собственный интерфейс для общения с пользователем, который вынужден будет использовать для каждой программы собственный набор команд. Собственная структура команд в каждой программе — это примерно то же самое, что уникальный рулевой механизм или уникальная коробка передач в каждом отдельном автомобиле. Разнообразие, разумеется, хорошая вещь, но наличие стандартного интерфейса позволяет пользователю работать с разными программами в Windows с минимальными затратами времени на переучивание. Система Windows обеспечивает стандартный набор объектов пользовательского интерфейса, таких как окна, меню и пиктограммы. Доступный любой прикладной программе набор объектов пользовательского интерфейса означает сходность всех программ в Windows по внешнему представлению, что в свою очередь упрощает как обучение, так и пользование программами для Windows.

Система Windows обеспечивает многозадачный Графический Интерфейс Пользователя (Graphical User Interface - GUI), который просто "подталкивает" к написанию интерактивных программ. Система Windows представляет собой относительно новый тип операционной среды, оптимизированной для взаимодействия человека и программы. Программы для Windows по своей структуре отличаются от программ, предназначенных для работы в традиционных операционных системах. Их структура напоминает структуру программ, написанных для других систем с GUI, таких как Apple Macintosh и OS/2 Presentation Manager. Программы, работающие в этих системах, называются программами, которыми управляют события. Т. е. структура и сама работа этих типов программ ориентированы на генерируемые пользователем события (например, нажатия клавиш или кнопки мыши). Архитектура таких программ имеет больше общего с системным программным обеспечением и подпрограммами-обработчиками прерываний, нежели с традиционным прикладным программным обеспечением.

Традиционно прикладное обеспечение создавалось в виде программ с управляющей последовательностью, а не программ, управляемых событиями, т. е. сама программа диктовала пользователю последовательность действий для достижения конечной цели. В противоположность этому программы, управляемые событиями, позволяют пользователю в процессе выполнения задачи самому диктовать программе те или иные предпринимаемые шаги. Переход от программ с управляющей последовательностью к программам, управляемым событиями, требует нового образа мышления. Данная книга поможет Вам в этом.

История Windows

Все системы с GUI берут начало от разработки, выполненной фирмой Херох. В 1970 г. фирма Херох основала Исследовательский Центр Пало Альто (Palo Alto Research Center - PARC). Его целью являлось создание новой архитектуры в подходе к обработке информации. Помимо прочего, PARC занимался лазерной печатью, локальными сетями, графическим интерфейсом пользователя и объектно-ориентированным программированием.

На базе этих исследований было построено несколько версий машины, которая в PARC получила название "Alto". За несколько лет для внутренних исследователь-

ких работ были произведены и нашли широкое применение несколько сотен таких машин. По результатам исследований Alto фирма Херох создала коммерческую систему на базе GUI - рабочую станцию Star 8010. Фирма Херох представила Star в апреле 1981 г., на четыре месяца опередив IBM PC. Появившаяся новая многозадачная система поставлялась с мышью и графическим дисплеем с поточечной адресацией, на который выводились пиктограммы, окна и тексты с пропорциональным шрифтом. Хотя коммерческому успеху Star помешала ее высокая стоимость, она, тем не менее, явилась значительным достижением, будучи первой представленной на рынке системой на базе GUI.

Рассказывают, что примерно в 1979 г. Херох PARC посетил один из основателей Apple Computer Стив Джобс. Разнообразные системы Alto произвели на него настолько сильное впечатление, что, вернувшись в Apple, он повел исследования в направлении разработки аналогичной системы. Apple представила свою первую систему на базе GUI в 1983 г.: это была Apple Lisa. Затем фирма Apple выпустила следующую, Apple Macintosh. Для ее презентации Apple в январе 1984 г. купила рекламное время на соревнованиях "Super Bowl" и рекламировала Macintosh как компьютер, который спасет мир от кошмара "Старшего Брата", описанного в романе Джорджа Оруэлла "1984". Значение Apple Macintosh состоит в том, что это была первая система на базе GUI, имевшая коммерческий успех.

Фирма Microsoft начала работать над Windows весной 1983 г. Восемь лет прошло с тех пор, как основатели Microsoft Билл Гейтс и Пол Аллен написали интерпретатор Бейсика для первого в мире персонального компьютера MITS Altair. Два года прошло с начала выпуска персональных компьютеров IBM, которые поставлялись с двумя продуктами — Microsoft DOS и BASIC. Фирма Microsoft готовилась к выпуску версии DOS 2.0 с ее поддержкой иерархической файловой системы для жестких дисков другого нового продукта фирмы IBM — компьютера PC/XT.

В это время в фирме Microsoft ходили разговоры о создании системы на базе GUI для персональных компьютеров IBM, однако никаких конкретных планов не было. Основная причина состояла в том, что стандартная PC в те времена включала в себя два флоппи-диска, 64 Кбайта оперативной памяти и центральный процессор 8088. Было ясно, что для эффективной работы системы на базе GUI требуется значительно более мощное аппаратное обеспечение. Для быстрого доступа к данным нужен был жесткий диск, а для сложного графического программного обеспечения и графических данных, предъявляющих высокие требования к потребляемой памяти, такая система должна была иметь значительный объем доступной оперативной памяти. Однако вскоре произошло нечто, что подтолкнуло Microsoft к разработке GUI.

В феврале 1983 г. фирма VisiCorp, разработчик популярной в то время электронной таблицы VisiCalc, объявила о создании программного продукта GUI для IBM PC. Этот продукт "VisiOn" и послужил причиной разработки в Microsoft собственной системы на базе GUI. Главным образом речь шла о том, что в случае успеха VisiOn разработчики программного обеспечения отвернутся от стандарта MS DOS. А в Microsoft очень рано осознали программные стандарты и вопросы совместимости всегда будут критичными с точки зрения успеха в области микрокомпьютерной техники.

Была создана команда разработчиков, известная как "Группа Интерактивных Систем" (Interactive Systems Group - ISG), в нее входил бывший участник Херох PARC Скотт Мак-Грегор. Другой разработчик Windows, Нейл Концен, работал ранее над переносом электронной таблицы Multiplan фирмы Microsoft на Macintosh. Когда в ноябре 1985 г. появилась первая версия Windows, то она имела ряд свойств, отражавших влияние Херох PARC и Apple Macintosh. Однако все же это была собственная разработка Microsoft, обладавшая свойствами, по которым можно было предвидеть мощь еще не родившейся операционной системы OS/2.

Версия Windows 1.01 стала доступной на рынке программных средств в ноябре 1985 г. Эта первая версия обеспечивала автоматическое непересекающееся размещение программных окон на экране. Автоматическая организация окон ощутимо снижала трудоемкость с точки зрения пользователя. Первая версия также поддерживала

перекрывающиеся, или "всплывающие" окна, служившие главным образом для создания в них блоков диалога. Windows версии 1 поддерживала "тостер для трех гре-нок", обеспечивающий доступ к системному меню. Разработчики шутили, что это была небольшая вентиляционная отдушина, благодаря которой экран не "перегревался от скорости графики".

Первая версия Windows была предназначена для работы на IBM PC с двумя флoppi-дисками, 256 Кбайт оперативной памяти и процессором Intel 8088. Совершенно случайно данная конфигурация совпала с той, которая использовалась самой командой ISG на ранних этапах разработки Windows. Те средства, которые сегодняшние разработчики считают вполне обычными, появились намного позже, речь идет о жестких дисках и быстродействующих сетях с файл-серверами и принт-серверами. Хотя использовавшиеся при разработке примитивные аппаратные средства снижали темп работ, в Microsoft понимали, что именно такое оборудование, скорее всего, будет у разработчиков прикладных программ для Windows. Разработка в такой среде помогла добиться максимальной пригодности Windows для дальнейшего использования прикладными программистами.

Следующим важным шагом в развитии Windows стала версия 2 (сентябрь, 1987 г.). Система Windows 2 работала уже с перекрывающимися окнами. Но конечных пользователей не устроило автоматическое непересекающееся размещение окон, в результате чего от него пришлось отказаться и направить свои усилия на обеспечение соответствия Windows другой графической среде OS/2 Presentation Manager, которая была объявлена в апреле этого же года.

Схожесть пользовательских интерфейсов Windows и OS/2 позволяет легко перейти от одной системы к другой. Фактически пользовательский интерфейс, используемый этими двумя операционными средами, является частью более крупной стратегии IBM по созданию непротиворечивости программных средств, которая распространяется как на самые малые персональные компьютеры, так и на самые большие. Эта стратегия называется Архитектура среды для разработки приложений (Systems Application Architecture — SAA). Основное назначение SAA состоит в том, чтобы программу, написанную на одной платформе, можно было легко перенести в другую рабочую среду. Например, при полной реализации SAA в один прекрасный день OS/2 появится на больших машинах IBM, на мини-компьютерах и, разумеется, на персональных. Частью SAA, касающейся вопросов пользовательского интерфейса, является Система пользовательских интерфейсов (Common User Access - CUA).

Помимо нового пользовательского интерфейса, одним из ключевых улучшений Windows 2 явилось усовершенствование использования памяти в форме поддержки расширенной памяти - памяти, доступной посредством Спецификации Дополнительной (Отображаемой) Памяти (Expanded Memory Specification - EMS). EMS описывает метод переключений между банками памяти, позволяющий работать с дополнительной памятью, несмотря на то что одновременно они физически недоступны. В Windows 2 EMS-память представляет возможность одновременно иметь в оперативной памяти большее количество программ Windows, поскольку каждой из них назначался собственный банк EMS-памяти. EMS-память отчасти решила проблему резкой нехватки доступной памяти, однако далеко не полностью, поскольку Windows 2 работала только в реальном режиме. Даже на более мощных процессорах 80286 и 80386 фирма Intel по причинам совместимости оставила ограничение на 1-мегабайтное адресное пространство как и на более слабых процессорах 8088 и 8086.

Под звуки фанфар 22 мая 1990 г. Microsoft объявила рождение Windows версии 3 и начала ее коммерческое распространение. За шесть недель Microsoft продала 500 тыс. копий новой версии, побив все рекорды продаж программных продуктов за шестинедельный срок. Все наблюдатели отметили, что с коммерческой точки зрения Windows 3 имела потрясающий успех.

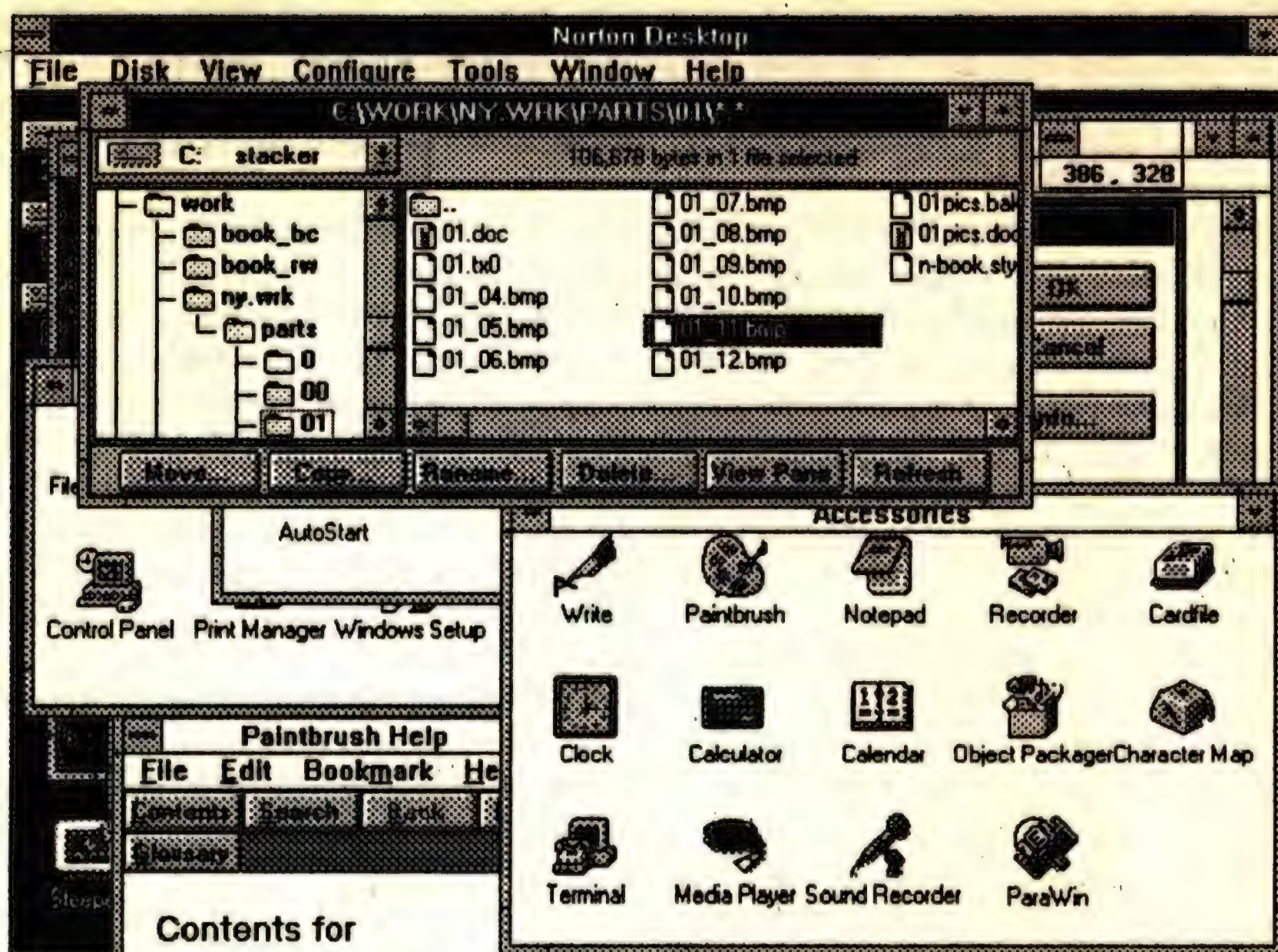


Рис.1.1. Система Windows версии 3.

Пользовательский интерфейс Windows 3 по сравнению с интерфейсом прошлых версий изменился, с учетом норм, ставших общепринятыми в 1990-х годах. Интерфейс имеет пропорциональный системный шрифт, придающий Windows более изящный вид и улучшающий читаемость текстов. Придающее ощущение трехмерности затенение изображения, цветные пиктограммы и разнообразные средства настройки экрана делают Windows более привлекательным для среднего пользователя. Windows 3 также лучше поддерживает запуск прикладных программ для DOS, что подсказало многим пользователям идею использования Windows в качестве первичного пользовательского интерфейса на компьютерах на базе DOS.

С точки зрения программирования Microsoft обеспечила весьма богатый выбор средств организации пользовательского интерфейса: предусмотрена поддержка меню, блоков списка и кнопок, предоставлена возможность в большей, чем раньше, степени управлять настройкой Windows. Меню в новой версии Windows могут быть вложены сколь угодно глубоко, а "отрывные" (с произвольной привязкой) меню дают программисту свободу выбора местоположения. Супервизор MS DOS, хорошо знакомый пользователям ранних версий, заменен набором программ, управляющих организацией программ и файлов, - Менеджером Программ (Program Manager), Списком Задач (Task List) и Менеджером Файлов (File Manager).

Наиболее важным внутренним свойством Windows 3 является поддержка расширенной (extended) памяти. В Windows 3 программы могут адресовать до 16 Мбайт оперативной памяти. При наличии процессора 80386 или старше Windows использует свойства управления памятью этих процессоров для организации виртуальной памяти. В улучшенном режиме процессора 386 (386-Enhanced) становится доступной виртуальная память, превышающая по объему почти в 4 раза физическую установленную память. Например, при 16 Мбайтах физической памяти (и достаточно большой доступной дисковой памяти) Windows обеспечивает 64-мегабайтное адресное пространство.

Windows 3, помимо прочего, имеет по сравнению с прошлыми версиями лучшую сетевую поддержку, упрощая подключение сетевых файл-серверов и серверов печати. Она поддерживает аппаратно-независимый растровый формат, обеспечивающий стандарт для совместного использования цветных битовых образов разными устройс-

твами. Если устройства, поддерживают более 256 цветов, прикладные программы имеют доступ даже к аппаратным цветовым палитрам. Это означает, что теперь Windows обеспечивает качество изображений, приближающееся к качеству цветных репродукций, как, например, в системах мультимедиа. Другим важным свойством является сложное встроенное средство формирования интерактивных подсказок, с помощью которого Вы можете обеспечить пользователя Вашей прикладной программы подсказками, выполненными по технологии "гипертекст".

Сразу после выхода этой книги в печать Microsoft начала представление новой операционной системы Windows NT (New Technology - Новая Технология). Эта система будет поддерживать несколько различных типов программ: программы для Windows 3.x, программы для MS DOS и программы с интерфейсом Posix (Posix - это Unix-подобный программный интерфейс). Также она будет представлять собой 32-разрядную версию Windows. Программы, написанные для Интерфейса Прикладных Программ (Application Programs Interface - API), получают доступ к новым средствам, таким как непрерывное адресное пространство, семафоры, цепочки задач и более мощный Графический Интерфейс Устройств (GDI). Рассмотрим некоторые аспекты, связанные с этой новой операционной системой.

Операционная система Windows NT

Похоже, что каждая фирма, разработавшая систему на базе GUI, создает затем ее "новую и улучшенную" версию. Фирма Xerox сначала разработала Alto, а затем улучшила ее, создав Star. Фирма Apple начала с Lise, а позднее разработала Macintosh. После создания Windows фирма Microsoft объединила свои усилия с IBM, чтобы создать OS/2 Presentation Manager.

Одна из целей разработки OS/2 состояла в том, чтобы дать возможность перенести сюда существующие программы, написанные для Windows. Для многих программ это удалось сделать, однако не для всех. В чем же причина? Хотя Presentation Manager по своей архитектуре очень близок к Windows, он реализует практически новый Интерфейс Прикладных Программ (API). Все имена функций у них отличаются. Например, Presentation Manager имеет функцию WinCreateWindow вместо функции Windows - CreateWindow. Параметры в целом похожих функций могут отличаться последовательностью задания или вообще отсутствовать. Presentation Manager вводит практически новый набор символьных констант. Если разработчик программы для Windows думал о переносимости кода в новую операционную систему, то для перехода на OS/2 потребуется та же работа, что и для перехода в любую другую среду GUI, например в Macintosh, X-Windows и т.п.

Операционная система Windows NT представляет собой не что иное, как признание Microsoft того, что она сделала ошибку с OS/2. Хотя чело Windows API украшают "бородавки", огромный объем наработанного программного обеспечения сделал эти "бородавки" родными. Несмотря на то что 32-разрядный интерфейс Windows API лечит некоторые болезни Windows, в целом он представляет собой интерфейс прикладных программ, унаследованный от 16-битового API из Windows 3.0. Новый API создавался с учетом вопросов совместимости и сохраняет те же имена функций, символических констант и структур данных, что и старый.

Единственное изменение, отличающие эти два интерфейса, состоит в том, что каждый 16-разрядный элемент теперь стал 32-разрядным. Хотя это изменение и может показаться радикальным, оно таковым не является. Далее Вы убедитесь, что каждый тип данных Windows определен с помощью переносимых типов. Другими словами, вместо short int или long int используются записываемые верхним регистром имена типов, например, HWND и HDC. При компиляции программы для любого другого API достаточно выбрать нужный включаемый файл для соответствующего API. Компилятор затем сам обрабатывает все различия между 16- и 32-разрядными значениями.

Что все это дает Вам как программисту, желающему писать под Windows? Начав использовать компилятор Borland C++ и библиотеки OWL для создания собственных

программ под Windows, Вы сможете почувствовать, что Вашим программам суждена долгая жизнь. Кроме того, программы под Windows 3.0 по двоичным кодам совместимы с Windows NT, даже если Windows NT работает на процессоре, не принадлежащем к семейству Intel. В настоящее время Windows NT параллельно разрабатывается для компьютеров на базе процессоров фирмы Intel и на базе процессоров MIPS RISC.

Теперь, зная происхождение Windows и представляя себе, в каком направлении происходит развитие этой системы, мы можем рассмотреть основные препятствия, с которыми Вы встретитесь во время обучения программированию под Windows. Важно понимать, что при этом Вам придется затратить немалые усилия, потому что учитывая гибкость пользовательского интерфейса, Вы можете прийти к ошибочному заключению, что с программным интерфейсом работать так же легко.

Действительно, проблема, с которой сталкивается программист, состоит в понимании фундаментальных принципов и моделей, включенных в архитектуру Windows. Проникнувшись "образом мышления Windows", Вы поймете, что программирование под Windows не сложнее любого другого рода программирования. Если же Вы раньше имели дело с другими системами на базе GUI, такими как Apple Macintosh, OS/2 Presentation Manager, или различными системами X-Windows, то при работе с Windows Вы найдете много общего с ними. Рассмотрим теперь собственно проблемы, которые могут возникнуть у Вас при изучении Windows.

Проблемы, связанные с программированием под Windows

Рассмотрим следующий сценарий. Днем в пятницу, выходя из офиса, Вы встретили своего босса. У него хорошие новости: одобрено Ваше предложение о разработке проекта под Windows. Это значит, что Вы и работающая на Вас команда программистов должны приступить к созданию первой в Вашей компании программы под Windows. Прежде всего, Вы должны позаботиться о том, чтобы все участники Вашей команды как можно быстрее овладели Windows. С какими проблемами столкнутся программисты, которые ранее программ под Windows не писали?

При условии, что программист профессионально работает на С, перед ним встают три основные проблемы: понимание концепции программы, управляемой сообщениями; управление графическим выводом; использование различных объектов пользовательского интерфейса, таких как окна, меню, блоки диалога и т.п. Если по какой-либо причине программист уже знаком с этими вопросами, то осваивать Windows ему будет проще.

Прежде чем перейти к подробному обсуждению перечисленных проблем, мы должны сделать одно предположение: допустим, Вы являетесь пользователем Windows, работающим в Windows полный рабочий день. Дело в том, что в пользовательском интерфейсе Windows имеются элементы, которые становятся очевидными только тому, кто имеет некоторый опыт работы с Windows в качестве конечного пользователя. Это, например, способы взаимодействия клавиатуры с мышью, способы работы с меню и кодовыми комбинациями клавиш, а также способы использования различных элементов управления в блоках диалога. Работая постоянно с Windows в качестве пользователя, Вы сможете лучше программировать под Windows. В крайнем случае, разыщите программу, с которой сможете работать на регулярной основе, это может быть текстовый редактор, пакет графики, программа эмуляции терминала или даже игра.

Рассмотрим три проблемы, с которыми сталкивается каждый программист, начинающий писать под Windows. В первую очередь, это проблема написания программ, управляемых сообщениями.

Проблема 1: Создание программ, управляемых сообщениями.

Большинство программистов привыкает создавать программы, в которых последовательность выполнения определяется самими процедурами. Такие программы имеют хорошо определенные начало, середину и конец. Рассмотрим, например,

программу, выводящую на дисплей последовательность экранов ввода данных для создания некоторого письменного документа, который может являться, допустим, авиабилетом или же товарной накладной. Блок-схема на следующем рисунке показывает строгую последовательность, в которой может быть выполнена такая программа.

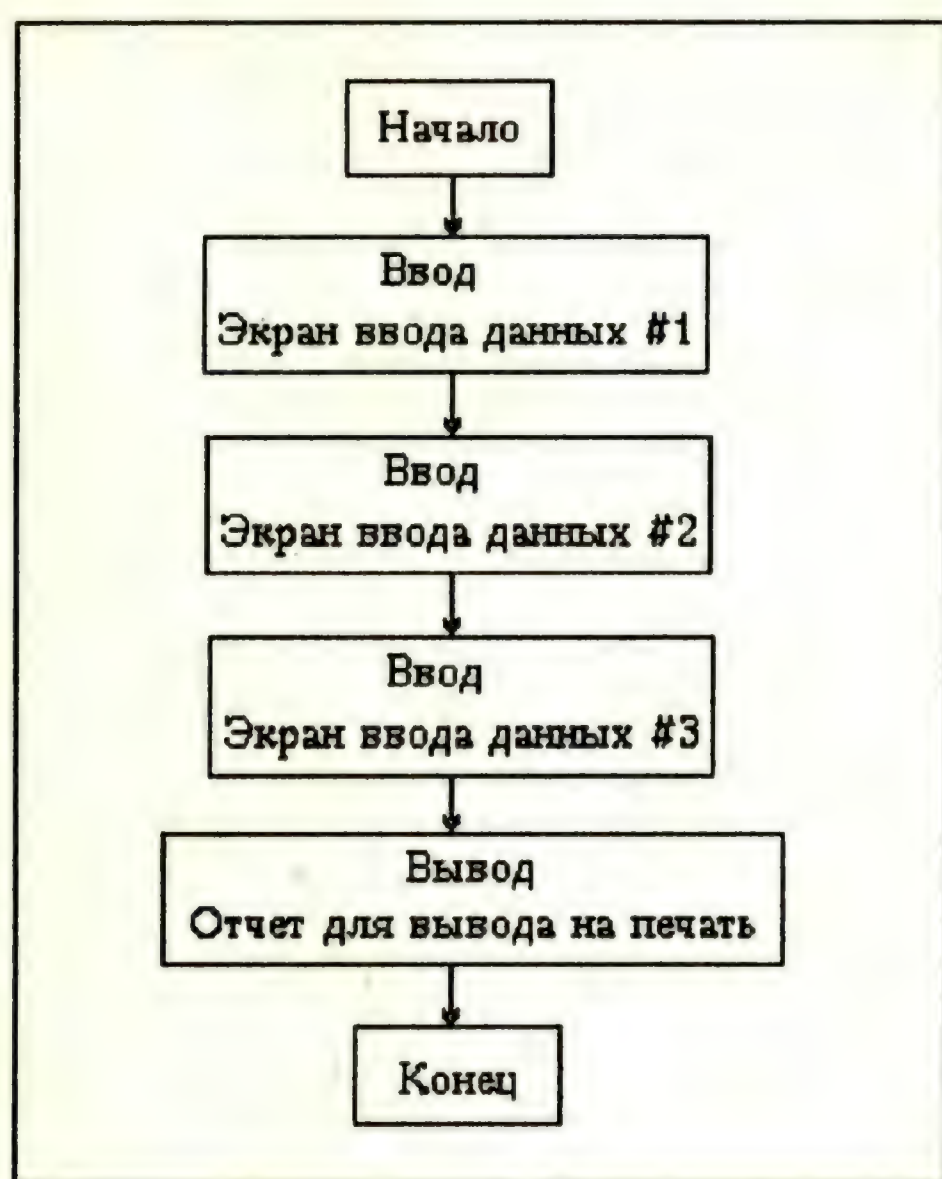


Рис.1.2. Блок-схема программы с жесткой управляющей последовательностью.

Предположим, что такая программа используется в авиакассе для оформления авиабилетов. Первый экран ввода данных принимает информацию о пассажире: имя, адрес и т.д. Второй экран позволяет вводить информацию о рейсе - тарифы и время отправления. И, наконец, третий экран принимает информацию о заплаченной пассажиром сумме на основании тарифов, выведенную на предыдущий экран. Каждый экран ввода данных должен быть правильно заполнен, прежде чем кассир сможет перейти к следующему шагу, а для того чтобы правильно был отпечатан билет, необходим ввод точной информации на всех трех экранах.

На первый взгляд кажется, что такой способ работы программы вполне приемлем. И действительно, основная работа такой программы состоит не просто в оформлении билета, но и в обеспечении ввода правильной информации о пассажире и проверки того, что полученная сумма соответствует действующим тарифам. Однако такой подход имеет ряд ограничений, вытекающих из природы программ с жесткой управляющей последовательностью.

Например, поскольку программа сама диктует последовательность операций, кассир не может перейти ко второму экрану о тарифах и рейсе, пока не завершит ввод информации о пассажире. Хотя такой подход позволяет кассиру ничего не упустить, объективным его результатом является создание ненужных шагов.

Дело в том, что, требуя обязательного ввода всей нужной информации, такая программа не принимает во внимание постоянно встречающиеся в реальном мире исключения из правил. Например, при необходимости оформить группу билетов для отправляющейся на каникулы семьи, кассир должен для каждого билета снова пройти по всем трем экранам ввода данных. Всякий раз при этом программа потребует обязательного ввода всей необходимой ей информации, но за счет дополнительных действий кассира.

С другой стороны, программа, управляемая событиями, позволяет кассиру вводить данные в той последовательности, в которой это окажется удобнее. Может случиться так, что кассир выберет последовательность, совпадающую с предлагаемой обычной программой. Однако у него всегда будет возможность выполнять все необходимые действия в последовательности, диктуемой конкретными требованиями клиентов. На следующем рисунке приблизительно показано, каким образом программа, управляемая событиями, может заменить традиционную программу с управляющей последовательностью, описанную выше.

Кроме того, это всего лишь один аспект, отличающий программу, управляемую событиями, от программы с управляющей последовательностью. Операционная система, управляемая событиями, такая как Windows, идет еще дальше, например, предоставляя кассиру в пределах экранов ввода данных колоссальную степень гибкости с точки зрения последовательности выбора заполняемых полей.

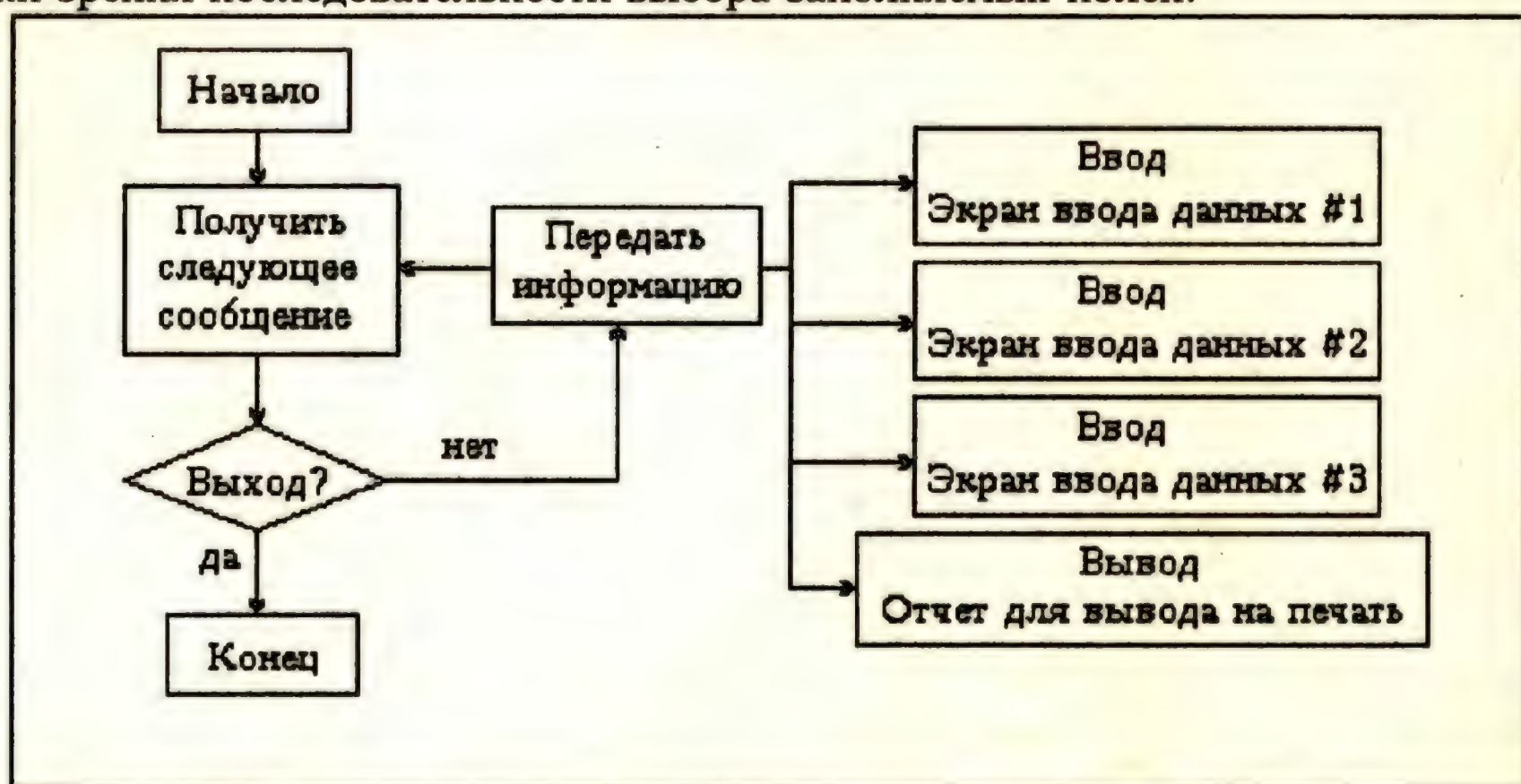


Рис.1.3. Программа, управляемая событиями.

Программа с управляющей последовательностью построена на негибко организованном наборе режимов. Режим - это состояние программы, в котором действия пользователя интерпретируются некоторым конкретным образом, производя конкретный набор результатов. О такой программе программисты, привыкшие работать с GUI, могут сказать, что работать с режимами неудобно. К сожалению, даже это высказывание будет большим упрощением.

Основная проблема с режимами возникает в связи с тем, что пользователь не может достаточно просто переходить от одного режима к другому. В нашей рассматриваемой программе, построенной по принципу управляющей последовательности, режимом является каждый из четырех описанных шагов. И поскольку программа диктует пользователю строгую последовательность действий, он не имеет возможности самостоятельно выбрать структуру использования программы, которая отражала бы различные возникающие при работе потребности.

Другая проблема с режимами возникает в программах, где пользователь должен сам помнить, в каком именно режиме он находится. По-хорошему, программа обязана обеспечивать некоторые визуальные ключи, помогающие пользователю идентифицировать текущий режим, в котором находится программа. Система Windows имеет множество объектов пользовательского интерфейса, предназначенных для этого. Например, форма курсора мыши может указывать, когда графический пакет находится в режиме вычерчивания прямоугольников, а когда - в режиме помещения на экран текстов. Далее Вы увидите, что модальные блоки диалога представляют собой очень распространенный способ приема от пользователя информации о режиме, позволяющей выполнить команду.

Режимы в программе должны разрабатываться очень тщательно, чтобы предотвратить потерю данных при ошибочном входе пользователя в тот или иной режим. В число популярных рассказов на эту тему входит история о текстовом редакторе Bravo, который был написан в Хегох PARC в 1970-х годах. В этом редакторе команды были назначены на обычные клавиши клавиатуры. Например, буква "i" переводила редактор в режим вставки символов в текст, "d" - в режим удаления и т.д. Однажды пользователь хотел ввести в документ слово "edit" ("редактировать"), но забыл войти в режим вставки. Редактор интерпретировал выполненные нажатия клавиш следующим образом:

E(verything)	выбрать весь документ целиком
D(elete)	удалить
I(nsert)	войти в режим вставки символов в текст
T	набрать букву "t"

Увы... Все содержимое документа заменилось на одну букву "t". При разработке пользовательского интерфейса Вашей программы Вы должны помнить о режимах и встраивать защитные средства от такого рода неприятных сюрпризов.

С точки зрения программирования модальные программы легче реализовать, чем немодальные. Код, поддерживающий каждый конкретный режим, можно писать и отлаживать изолированно от остальных частей программы. Система Windows упрощает написание немодальных программ, поскольку все взаимодействие с внешним миром, все события передаются программе не модальным способом. Сообщения генерируются всеми событиями, представляющими для программы интерес.

Что такое сообщение? Это информация о некотором изменении в пользовательском интерфейсе, например, перемещении окна или нажатии клавиши на клавиатуре. Сообщения уведомляют программу, что заданный отрезок времени истек. Они применяются для операций с совместным использованием данных.

С точки зрения программирования сообщение представляет собой 16-битовое значение без знака, которому для простоты чтения назначена символьная константа, начинающаяся с символов WM_ (сокращение для "Window Message" - "Сообщение Окна"). Например, сообщение WM_LBUTTONDOWN говорит программе, что пользователь нажал левую кнопку мыши. Другое сообщение - WM_LBUTTONUP - передается после отпускания левой кнопки мыши. В этой книге мы будем описывать различные сообщения в контексте рассматриваемых событий, к которым они относятся. Если же Вам не терпится увидеть все типы сообщений системы, Вы можете заглянуть в приложение A, в котором сводятся различные типы сообщений Windows.

Сообщения очень важны с точки зрения пишущего для Windows программиста. Большая часть Вашей работы в этом качестве будет заключаться в принятии решений относительно того, какие сообщения будут обрабатываться Вашей программой, а какие игнорироваться. Важно помнить, что сообщения появляются не в каком-то жестко заданном порядке. Если Вы привыкли к традиционным программам, то сначала такой подход может показаться дезорганизирующим и хаотическим. Будет такое ощущение, что сообщения летят к Вам, словно пули. Чтобы помочь понять прохождение сообщений по системе, фирма Borland предлагает программу WinSight, которая позволяет "изнутри" понаблюдать за сообщениями Менеджера Программ (Windows Program Manager).

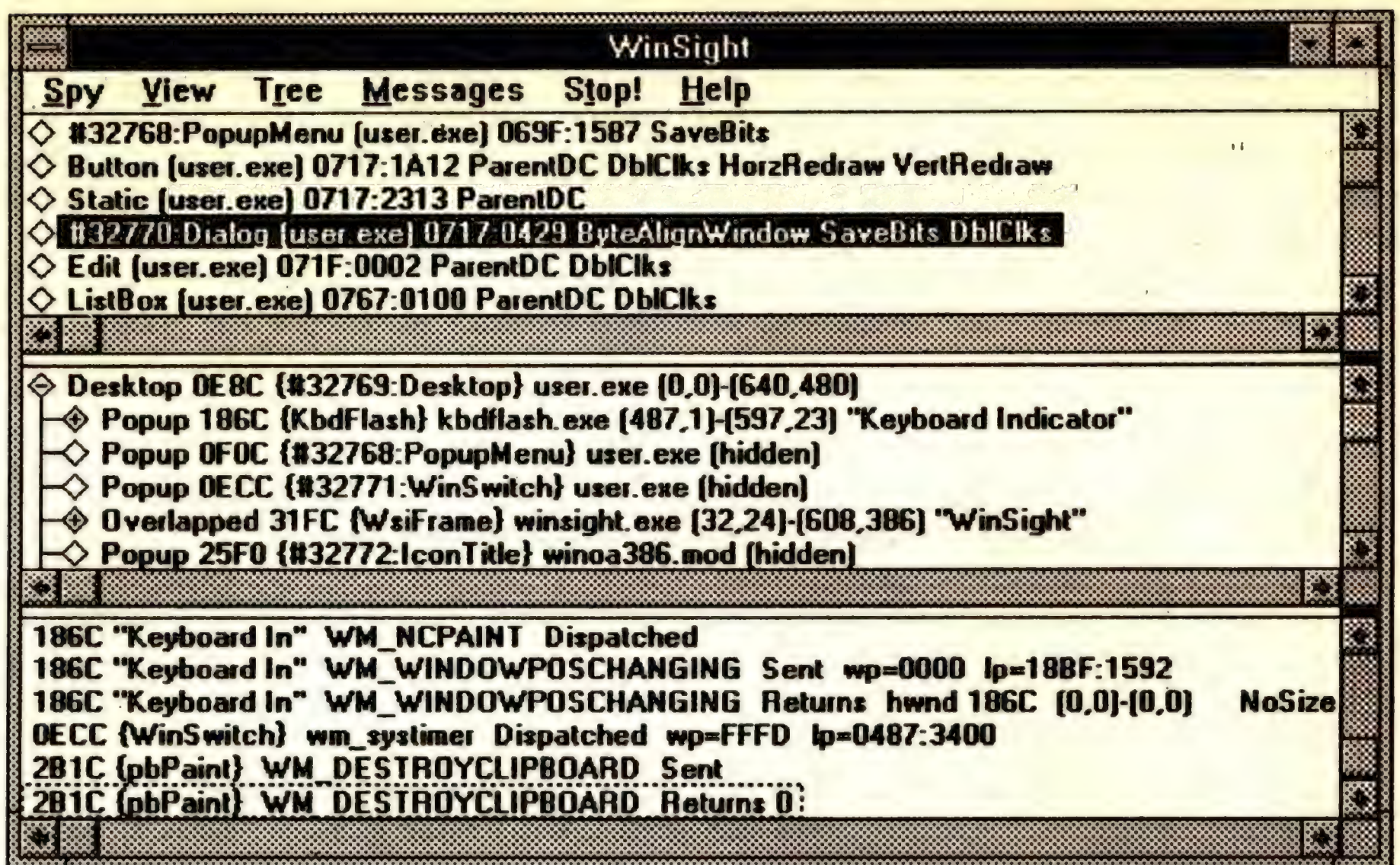


Рис.1.4. Использование программы WinSight для просмотра сообщений Program Manager.

Принятая в Windows ориентация на сообщения лучше всего подходит для программ, требующих интерактивности высокой степени. Поэтому компиляторы языков программирования, имеющие малую степень интерактивного взаимодействия с пользователем, практически ничего не выигрывают, будучи реализованы как программы под Windows. В отличие от них, игры и текстовые редакторы очень хорошо приспособлены для Windows, поскольку и те, и другие обладают высоким уровнем взаимодействия с пользователем. Электронные таблицы и программы для ввода данных также являются явными кандидатами на реализацию их в Windows в качестве интерактивных программ, управляемых событиями.

В такой управляемой событиями операционной системе, как Windows, высокий приоритет имеет возможность вмешательства со стороны пользователя в любой точке процесса. Напротив, в программах с управляющей последовательностью высший приоритет имеет именно принудительно задаваемый порядок выполнения действий. В таких программах специалисту крайне затруднительно создавать произвольные правила относительно порядка предпринимаемых шагов. Хотя программы с управляющей последовательностью можно создавать и в Windows, необходимые для этого дополнительные затраты труда программиста гарантируют, что такого рода ограничения будут создаваться только там, где это действительно необходимо.

Сообщения и планирование программ

Система Windows является многозадачной системой. Для пользователя это означает, что несколько программ может выполняться одновременно. Разумеется, при одном процессоре физически одновременное выполнение программ невозможно. Вместо этого происходит поочередное переключение с одной программы на другую. В традиционных операционных системах такое планирование выполняется по системным часам. Каждой программе назначается "квант времени", в течение которого она может выполняться. По истечении этого кванта программа прерывается и запускается другая. Это называется планированием задач с вытеснением. При планировании с вытеснением прерывание программ выполняется операционной системой.

Система Windows не поддерживает планирование с вытеснением. Windows имеет систему планирования задач без вытеснения. Вместо прерывания программ операци-

онной системой каждая программа по собственной инициативе прерывает свое выполнение, разрешая работать другим программам.

Система планирования Windows встроена в ее механизм доставки сообщений. Когда программа закончила обработку одного сообщения, она запрашивает следующее. Система планирования на базе сообщений означает, что пользователь является конечным источником принятия решений о том, какая задача должна выполняться в следующий момент. Если пользователь желает работать с некоторой программой, он фокусирует свое внимание на одном из окон программы, например, нажимая над ним кнопку мыши или выбирая это окно с помощью клавиатуры. Каждое из этих действий вызывает передачу программе сообщений, которые и предоставляют ей "квант времени", необходимый для выполнения.

Сообщения обеспечивают программу входными данными, но это только половина дела. Вторая половина относится к производимому программой выводу. Вывод в Windows означает всегда одно и то же: графический вывод. Это вторая проблема, с которой столкнется новичок в Windows и которая является предметом нашего дальнейшего обсуждения.

Проблема 2: Графический вывод.

Весь создаваемый программами для Windows вывод является графическим. На рис.1.7 показаны примеры линий, закрашенных фигур и текстов, которые обеспечивает GDI. Программисты, привыкшие к среде, ориентированной на текстовый вывод, обнаружат, что графический вывод требует от них нового образа мышления. Как и следовало ожидать, графический вывод означает возможность рисования на экране геометрических фигур - линий, окружностей, прямоугольников и т.п. Кроме того, и текст здесь рассматривается как графический объект. Это упрощает чередование на экране текста и геометрических фигур. Как это ни странно, упрощая вывод графических фигур, такого рода системы делают более сложным текстовый вывод.



Рис.1.5. Примеры линий, закрашенных фигур и текста, выводимых с помощью интерфейса графических устройств (GDI).

Геометрические формы вывести легче потому, что Вашей программе при этом не приходится вычислять каждый пиксель. Например, с помощью простого вызова подпрограммы Rectangle GDI позволяет Вашей программе нарисовать закрашенный прямоугольник. Текстовый вывод сложнее, поскольку графическая ориентация GDI требует обращения с текстом как с графическим объектом. Текст позиционируется по координатам пикселей, а не по позициям знакомест символов на экране.

Аппаратно-независимая графика

GDI обеспечивает аппаратно-независимую графику. Это означает, что программа для Windows может выполнять вывод на любое устройство с помощью одного и того же набора вызываемых подпрограмм. Например, подпрограмма `Rectangle` вызывает-ся для прорисовки прямоугольников как на экране дисплея, так и на принтере. GDI выполняет вместо Вас большой объем работы, так что с точки зрения программы все устройства выглядят одинаково. Сюда входят как устройства, которые только зажигают и гасят пиксели, например дисплейный адаптер (CGA), так и весьма "интеллектуальные" устройства, которые выполняют сложные графические операции, например принтеры PostScript. Каждое устройство имеет драйвер, отвечающий за фактическое выполнение графического вывода. Для устройств, которым при этом необходима помощь, GDI обеспечивает программную эмуляцию, использующую для реализации функций высокого уровня средства низкого уровня этого устройства.

GDI знает о четырех типах устройств: экране дисплея, устройстве для получения твердых копий (принтеры и плоттеры), битовых образах, а также метафайлах. Физическими устройствами из них являются два — экран дисплея и устройство для получения твердых копий. Два остальных — битовые образы и метафайлы — являются псевдоустройствами. Псевдоустройства обеспечивают средства для хранения изображений в оперативной памяти или на диске, а также стандартный способ совместного использования графических образов различными прикладными программами.

При выводе информации на экран дисплея GDI обеспечивает оконно-ориентированную графику. Оконно-ориентированная графика означает следующее. Каждое окно рассматривается как отдельная область прорисовки. Когда программа выполняет в окне прорисовку, координаты по умолчанию устанавливаются таким образом, чтобы точка начала координат (0,0) находилась в верхнем левом углу клиентной области окна.

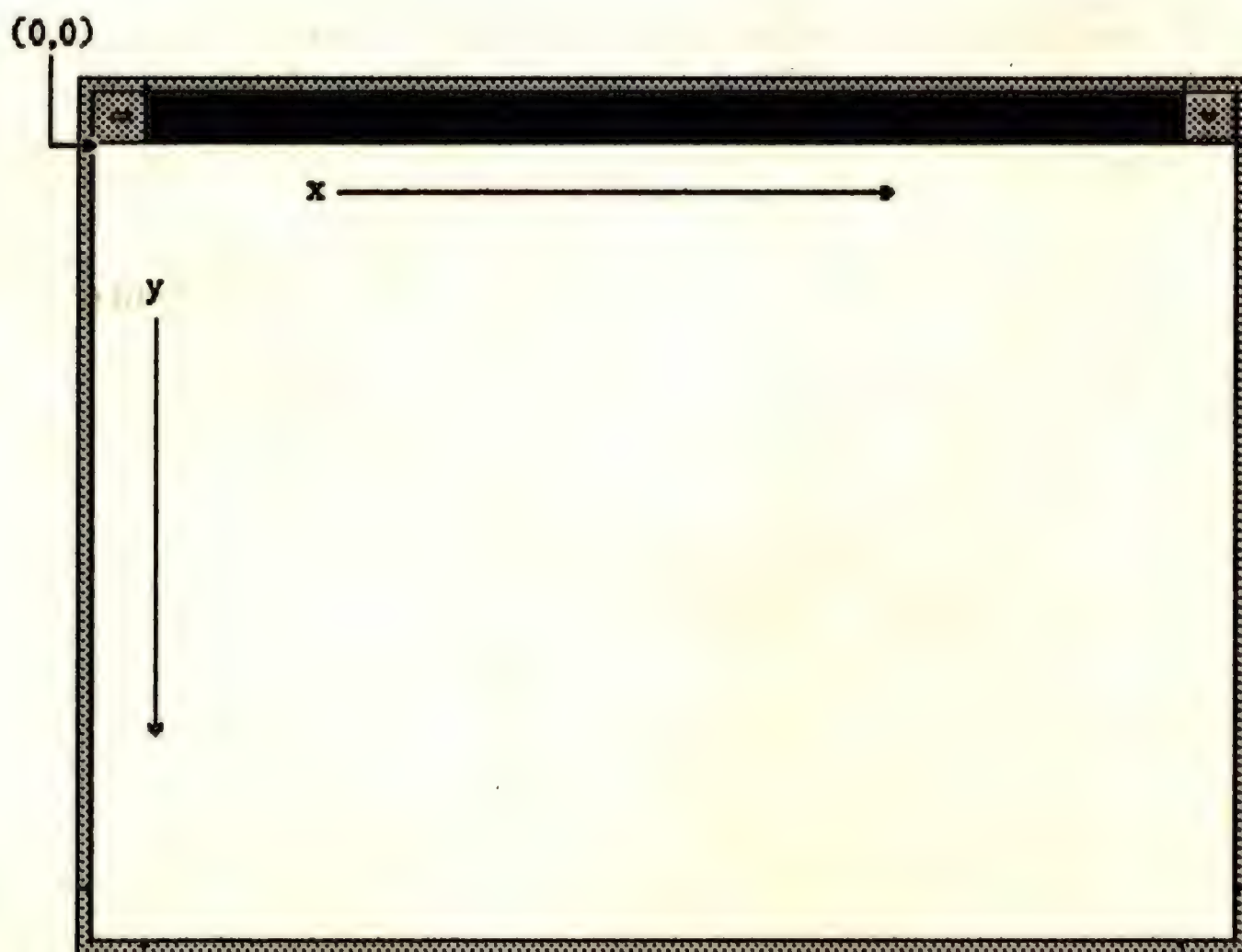


Рис. 1.6. Используемое по умолчанию начало координат клиентной области.

Оконно-ориентированная графика также означает, что рисунки автоматически отсекаются по границам окна. Отсечка означает, что прорисовка в каждом окне ограничивается областью окна. Даже если окно попытается нарисовать что-либо вне своих границ, ему это не удастся. Таким образом окно автоматически защищено от любых посторонних пикселей, рисуемых другими окнами. Подобный механизм защиты

работает двусторонне, так что при выводе в текущее окно Вам не нужно заботиться о том, чтобы случайно не затереть окно другой программы.

Проблема 3: Объекты пользовательского интерфейса.

Система Windows имеет встроенную поддержку ряда объектов пользовательского интерфейса: окон, пиктограмм, меню, блоков диалога и т.п. Встроенная поддержка означает, что затраты труда на создание и организацию этих объектов с Вашей стороны минимальны. В частности, если бы Вы сами захотели написать подпрограммы для реализации этих объектов, это заняло бы у Вас массу времени. При этом в результате Вы получили бы менее гибкие и менее защищенные от ошибок объекты, нежели предоставляемые Вам системой Windows.

Чтобы воспользоваться всеми преимуществами объектов пользовательского интерфейса, Вы должны понимать, каким образом каждый из них реализован. Рассматривая различные типы объектов пользовательского интерфейса, мы заглянем "внутрь" устройства и попробуем определить способы конкретной реализации каждого из них. Во многих случаях это может означать обсуждение сообщения, связанного с данным объектом пользовательского интерфейса. В других случаях придется углубиться в различные библиотечные подпрограммы Windows, управляющие тем или иным типом объекта. Сейчас же просто опишем объекты и их роль в пользовательском интерфейсе.

Среди всех объектов пользовательского интерфейса наиболее важными являются окна. Любая программа, предназначенная для взаимодействия с пользователем, должна иметь окно, поскольку именно через окно она будет принимать ввод с мыши и клавиатуры и выполнять вывод на экран из программы. Все прочие объекты пользовательского интерфейса, такие как меню, линейки прокрутки и курсоры, играют лишь вспомогательную роль по отношению к главному персонажу - окну.

Окно

Окно представляет собой самую важную часть пользовательского интерфейса. С точки зрения пользователя окно обеспечивает наблюдаемость некоторого объекта данных на экране компьютера. Однако этим дело не ограничивается, поскольку для пользователя окно отождествляется с самой прикладной программой. Когда пользователь запускает прикладную программу, он ожидает появления ее окна. Для завершения работы с программой пользователь закрывает соответствующее окно. Для выбора конкретной программы он выбирает относящееся к ней окно. В число стандартных элементов окна входят кнопка выбора системного меню, заголовок окна, кнопки максимизации (увеличения до размеров экрана) и минимизации (уменьшения до размеров пиктограммы) окна, клиентная (проблемная) область, рамка окна, а также полосы горизонтальной и вертикальной прокрутки.

Для программиста окно означает следующее. Оно служит для организации прочих объектов пользовательского интерфейса и направляет прохождение сообщений в системе, окно обеспечивает область экрана для связи с пользователем. Ввод направляется в данное окно и уже там становится доступным программе. Прикладные программы также используют окна, содержащие другие окна. Например, блоки диалога реализуются в виде набора небольших окон внутри более крупного окна.

Каждое окно принадлежит классу окон. Класс окон — это шаблон, по которому реализуются реальные окна. С каждым классом окон и, следовательно, с каждым окном связан специальный тип подпрограммы, называемый процедурой окна. Задача процедуры окна состоит в обработке поступающих окну сообщений. В такой ориентированной на сообщения операционной системе, как Windows, это, естественно, является важной задачей.

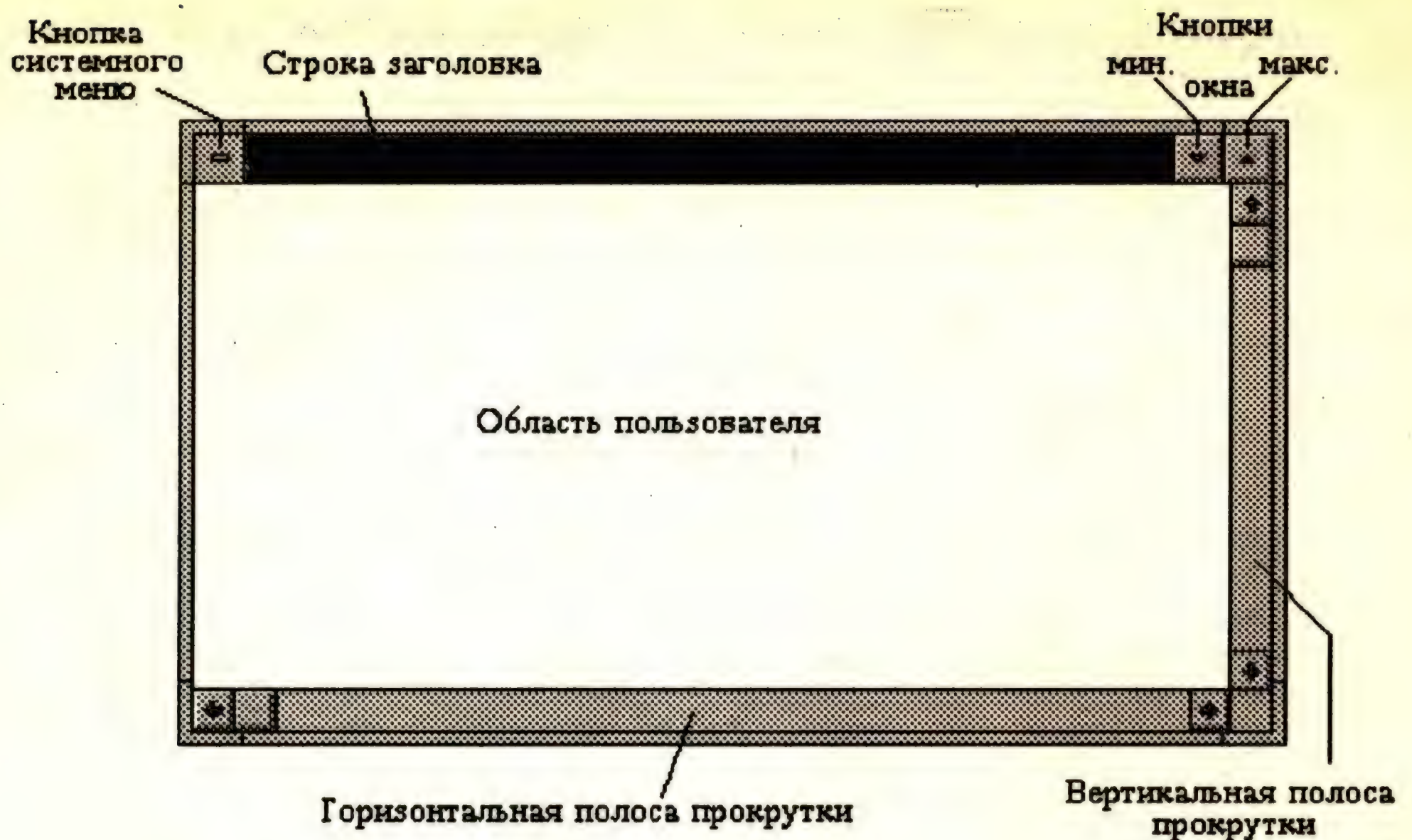


Рис.1.7. Стандартные участки окна.

Действительно, как программист Вы большей частью будете принимать решения о том, как обрабатывать те или иные сообщения, принимаемые процедурой окна. Процедура принимает направляемый окну ввод с мыши или клавиатуры, поступающий в виде сообщений. Она также принимает уведомления о других представляющих интерес событиях, таких как изменение размеров и положения окна на экране. В следующих нескольких главах мы прежде всего рассмотрим, каким образом поступают сообщения и как происходит их обработка процедурой окна.

Пиктограммы

Пиктограмма представляет собой небольшой рисунок, который служит для пользователя напоминанием о чем-либо. Системы на базе GUI построены на том принципе, что конкретное и видимое понятнее, чем абстрактное и невидимое. Пиктограммы обеспечивают конкретный, видимый символ, обозначающий команду, программу или некоторые данные. Делая их видимыми на экране, программа для Windows делает их доступными. Обеспечив видимость на экране всех возможных для выбора пользователем вариантов, программа уменьшает зависимость пользователя от того, насколько хорошо он запоминает информацию.

Чаще всего пиктограммой изображают доступную для выбора программу. В окне Менеджера Программ такие пиктограммы напоминают пользователю обо всех программах, которые он в текущий момент может запустить. На рабочей экранной панели пиктограммы напоминают пользователю о текущих работающих программах, окна которых были закрыты. Пиктограммы могут также изображать команды. Например, программа Paintbrush пиктограммами показывает пользователю все имеющиеся в ней операции рисования и шаблоны закрашивания.

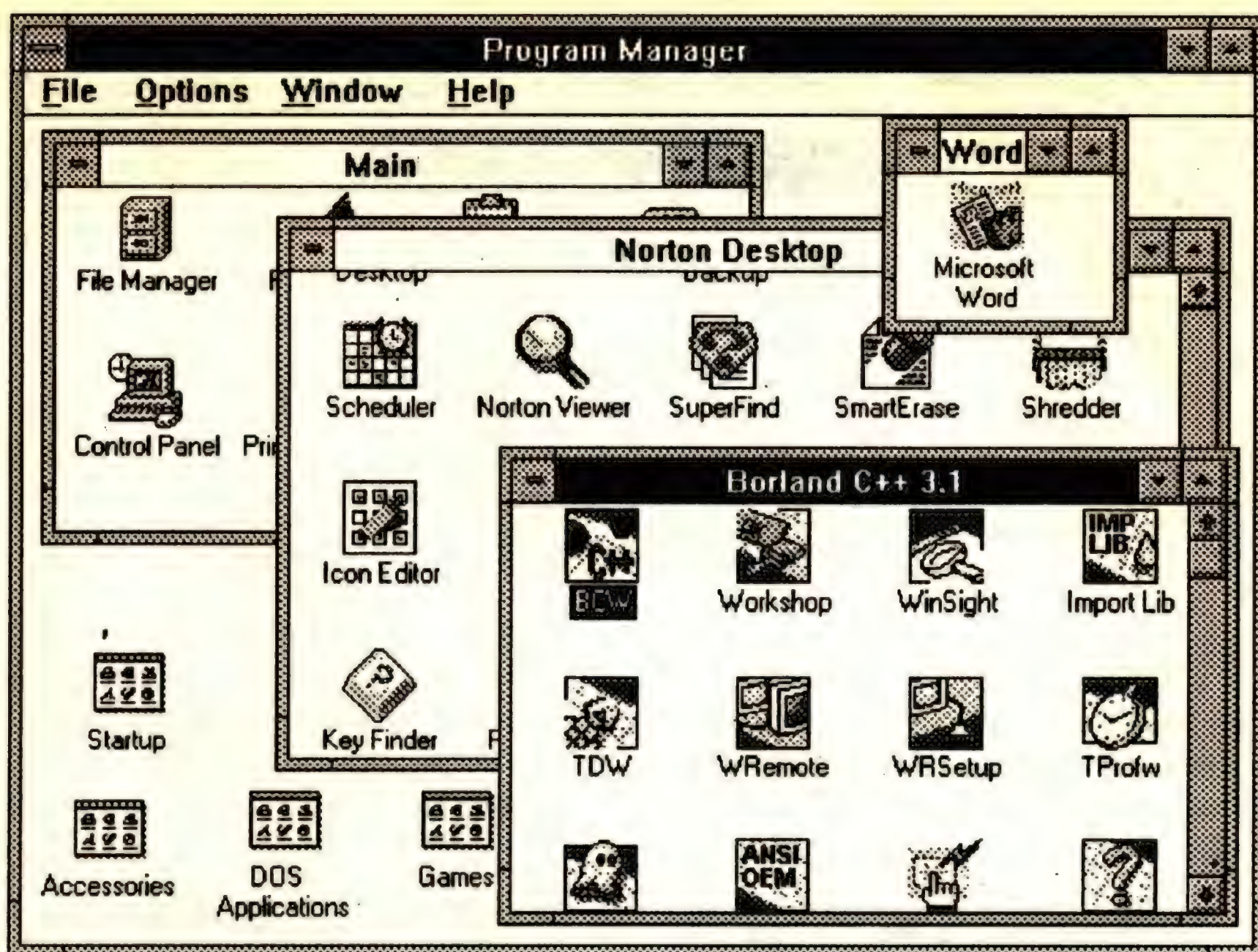


Рис.1.8. Пиктограммы, используемые программой Program Manager.

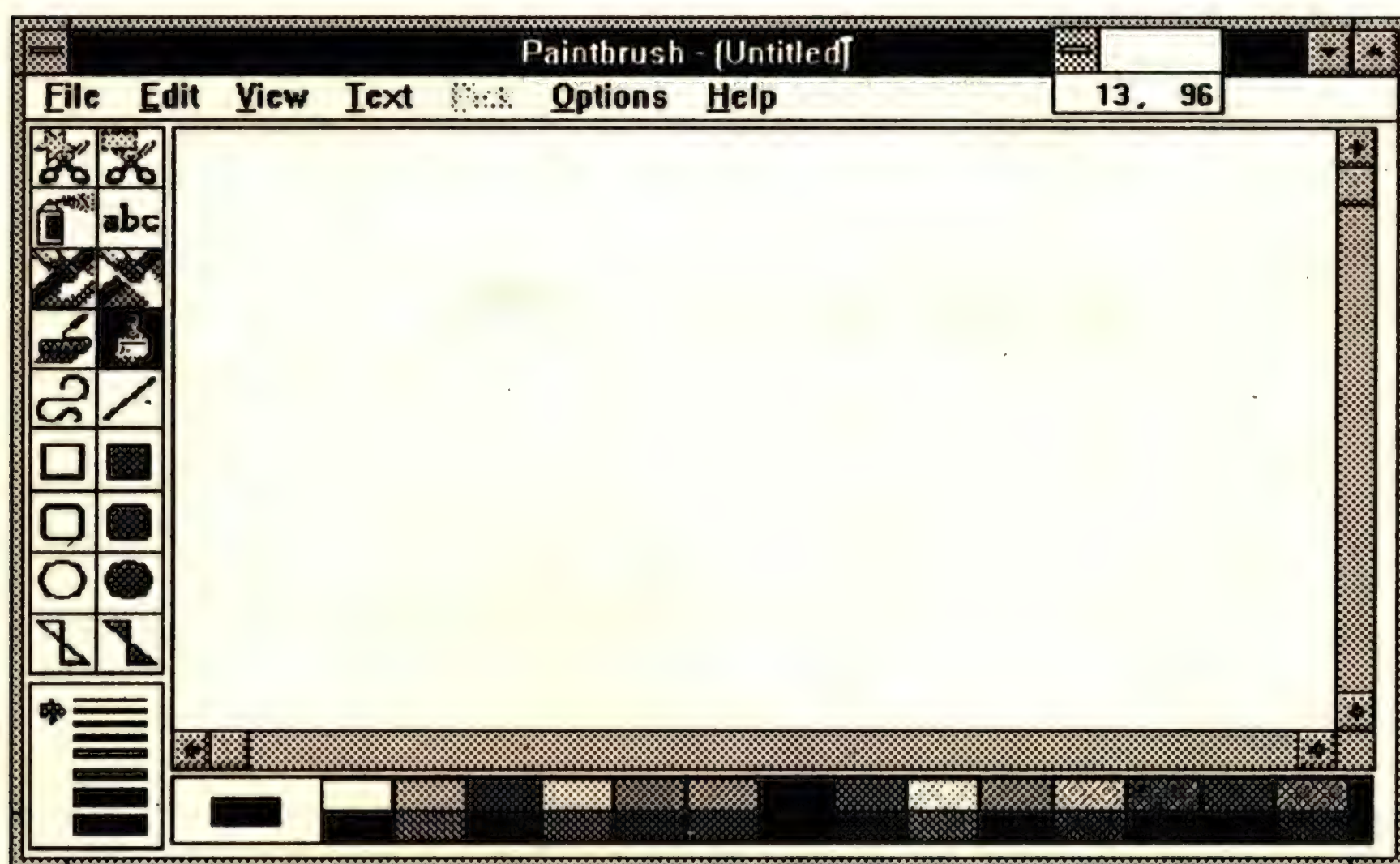


Рис.1.9. Пиктограммы, используемые программой Paintbrush.

Меню

Меню представляет собой список команд и функций программы. Windows имеет пять типов меню: системные, горизонтальные, выпадающие, вложенные и всплывающие. Системное меню обеспечивает стандартный набор операций, которые могут быть выполнены с окном. Эти операции еще называют системными командами. Пользователь обнаружит, что системное меню имеется в окне верхнего уровня каждой запускаемой им программы. Системные команды практически не требуют ника-

кого участия со стороны программы, поскольку Windows сама обеспечивает работоспособность и единообразие системных команд.

Move	
Size	
Migimize	
Maximize	
Close	Alt+F4
Switch To...	Ctrl+Esc

Рис.1.10. Системное меню.

Следующие три типа меню связаны между собой: горизонтальные меню привязываются к верху окна, выпадающие меню появляются при выборе соответствующих пунктов линейки меню, а вложенные — при выборе элементов выпадающих меню, помеченных стрелкой. Прикладная программа может вкладывать одно меню в другое до любого уровня вложенности, хотя в целом слишком глубокой вложенности рекомендуется избегать, поскольку это дезориентирует пользователя.

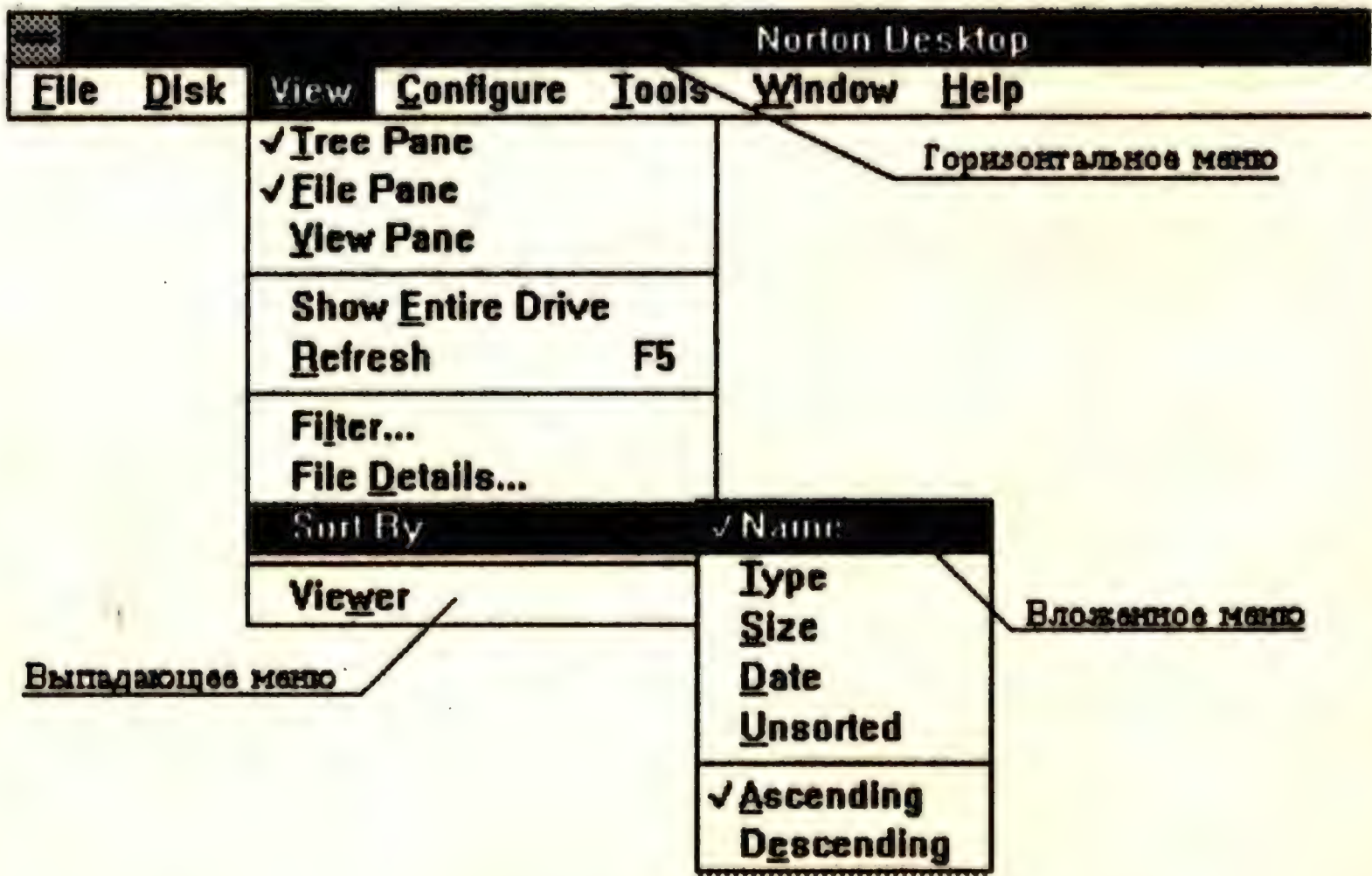


Рис.1.11. Три типа меню.

Всплывающие меню могут располагаться в произвольной позиции в окне и фактически в любой позиции на экране дисплея. Они предлагают альтернативу для программ, в которых нежелательно строить систему меню, привязанную к горизонтальному меню. Всплывающие меню могут появляться на экране, например, при нажатии кнопки мыши на некотором объекте или в ответ на нажатие комбинации "горячих" клавиш на клавиатуре.

U ndo	C trl+ Z
C ut	C trl+ X
P aste	C trl+ V
Copy To...	
P aste F rom...	

Рис.1.12. Всплывающее меню.

Полосы прокрутки

Если в окне показана полоса прокрутки, пользователь знает, что объект данных больше размеров окна. Полосы прокрутки позволяют пользователю управлять отображением таких объектов и видеть относительное положение отображаемой части объекта в целом. Полосы прокрутки бывают горизонтальными и вертикальными.

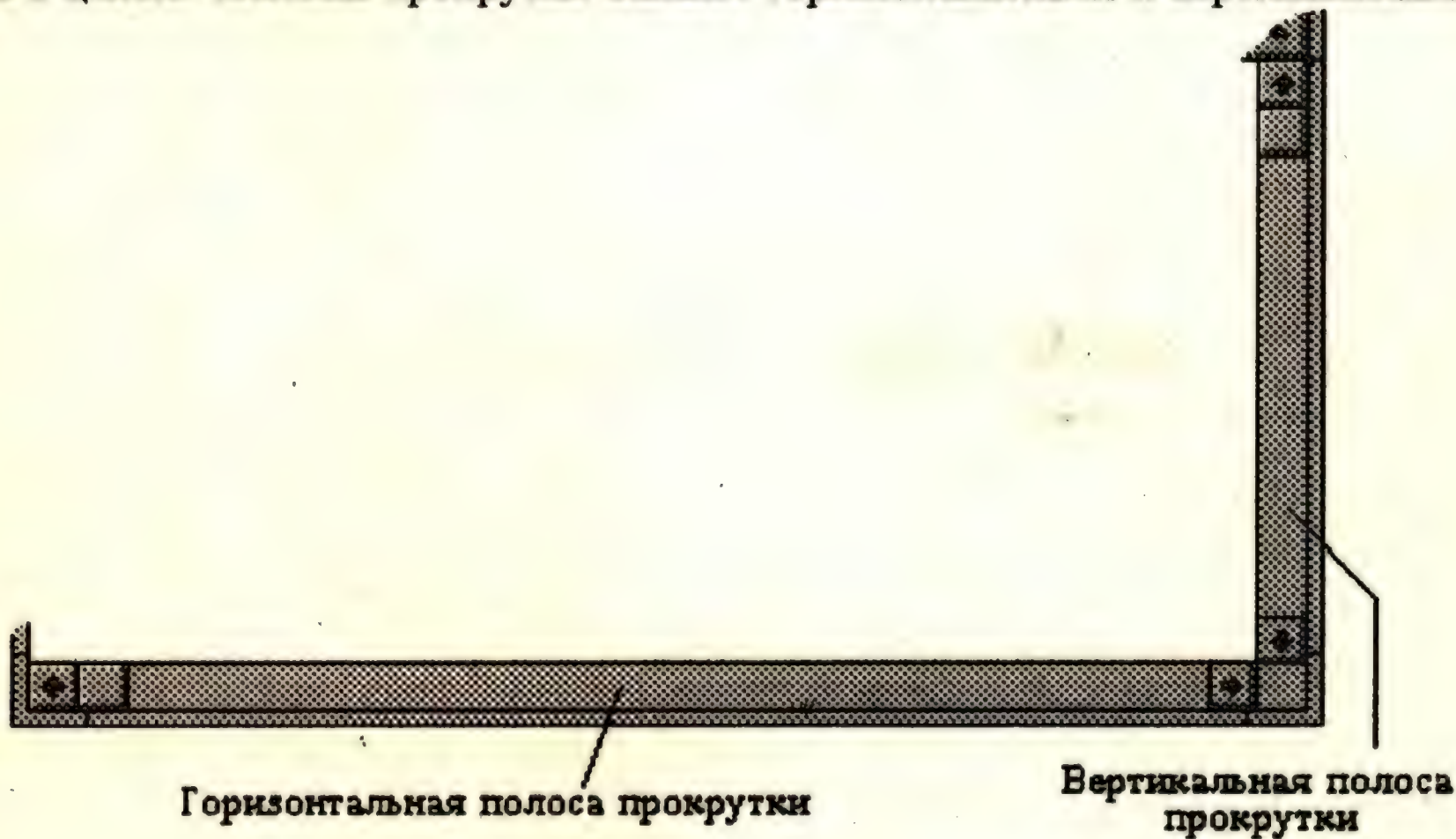


Рис.1.13. Вертикальные и горизонтальные полосы прокрутки.

Курсоры

Курсор представляет собой битовый образ, перемещаемый по экрану в ответ на перемещение мыши или другого координатного устройства. Программа может изменить форму курсора, чтобы отобразить некоторое изменение в системе. Например, часто программы выводят курсор в форме песочных часов, для указания того, что в это время происходит выполнение некоторого длительного процесса. Иногда программы формой курсора указывают на некоторый конкретный режим работы. Напри-

мер, программа для рисования может формой курсора обозначать тип рисуемого объекта.

С помощью утилиты Resource Workshop, входящей в состав компилятора Turbo C++, программист может создавать собственные курсоры. Курсоры можно создавать "на лету" - этот процесс описан в главе 16. Разумеется, Вы можете и обойтись без создания собственных курсоров, если те, что предлагает Windows, Вас устраивают.



Рис.1.14. Виды курсоров, используемые системой Windows.

Каретка

Каретка - это небольшая битовая матрица, которая является отметкой фокуса ввода с клавиатуры. Окно, управляющее вводом с клавиатуры (иначе называемое фокусом ввода), может создать каретку, чтобы сообщить пользователю о данном факте. Использовать каретку удобно по двум причинам. Во-первых, само название "каретка" удобно тем, что в большинстве других систем термин "курсор" закреплён за указателем позиции на экране, управляемым с клавиатуры. Однако в Windows курсором называется указатель мыши.

Во-вторых, с кареткой могут работать программы. Пользовательский интерфейс Windows поддерживает лишь одну каретку на экране. Следовательно, программы, в которых используется каретка, должны создавать ее при получении фокуса ввода с клавиатуры и удалять при потере фокуса. В главе 15, рассматривающей ввод с клавиатуры, мы увидим, что должна делать программа для правильной организации кареток.

Блоки диалога

Блоки диалога, иногда называемые просто диалогами, — это по сути стандартный способ приема программой ввода от пользователя. В частности, если пользователь ввел команду, требующую ввода дополнительной информации, блок диалога даст программе возможность принять эту информацию. При работе с программами для Windows в качестве части имени пункта меню часто можно видеть многоточие (...). Это многоточие указывает на то, что при выборе данного пункта меню Вам будет предложен дополнительный блок диалога.

Один из самых распространенных блоков диалога появляется, когда пользователь просит систему открыть файл. Это блок диалога для открытия файла, позволяющий ввести имя файла. В этом блоке также выводится два списка: один с именами файлов, а второй с именами каталогов и дисководов. Если пользователь не помнит конкретное имя файла, он может просматривать любые каталоги, пока не найдет там желаемое имя.

Отметим, что этот блок диалога содержит две экранные кнопки: одна называется "Ok" ("Подтвердить выбор"), а вторая - "Cancel" ("Отменить выбор"). Как правило, экранные кнопки в блоках диалога служат для запроса некоторого действия. В описанном блоке диалога таких возможных действий два. Экранная кнопка Ok говорит программе, что она может принять введенные пользователем значения. Напротив, кнопка Cancel заставит программу игнорировать значения, введенные пользователем в блоке диалога. В целом там, где это возможно, программа должна позволять пользователю отменить любой запрос без повреждения при этом файлов или данных в памяти.

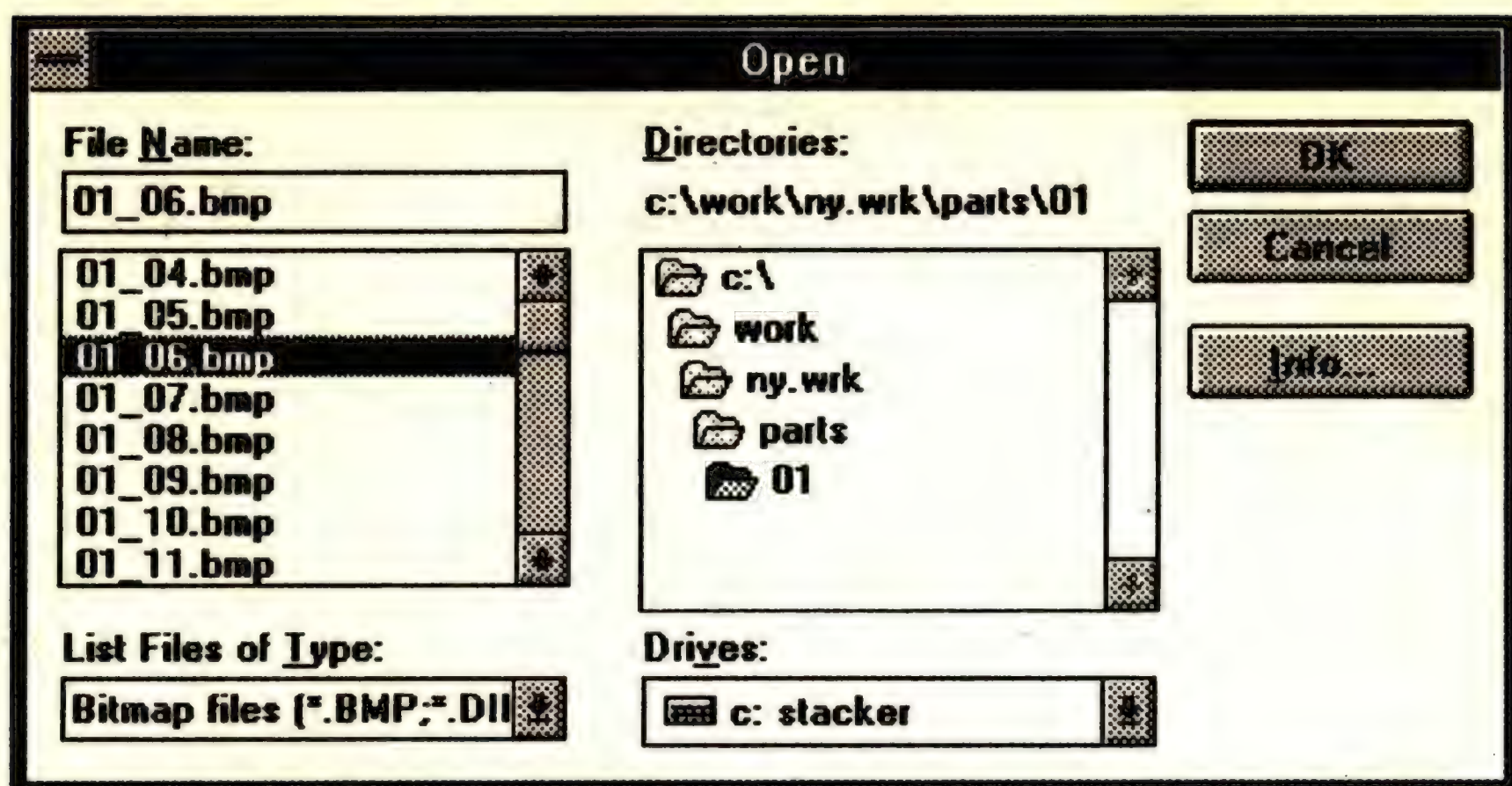


Рис.1.15. Блок диалога открытия файла.

Элементы управления блоком диалога

Как и все блоки диалога, диалог открытия файлов представляет собой окно, содержащее отдельные окна, которые либо выводят некоторую информацию, либо позволяют принять ввод от пользователя. Каждое из этих окон называется элементом управления диалогом. Например, диалог открытия файлов содержит девять элементов управления: две кнопки (Ok и Cancel), два блока списка, элемент управления редактированием и четыре статических текстовых элемента. Система Windows имеет шесть предопределенных классов окон, на базе которых создаются элементы управления блоками диалога: кнопки, комбинированные блоки, элементы управления редактированием, блоки списков, полосы прокрутки и статические элементы.

В главе 14 рассматривается создание блоков диалога. Вы увидите, как с помощью элементов управления можно организовать большую часть операций взаимодействия с пользователем. Например, элементы управления редактированием обрабатывают без Вашего участия все соответствующие нажатия клавиш на клавиатуре. Блок списков, которому передан некоторый список элементов, самостоятельно организует отображение на экране и просмотр с прокруткой всех элементов списка. Обратите внимание на то, что подобно многим другим частям Windows блоки диалога и их элементы управления в основном управляются событиями.

Подведем итог: для того чтобы научиться программированию под Windows, необходимо взять три концептуальных барьера - написание программ, управляемых сообщениями, управление графическим выводом и использование объектов пользовательского интерфейса. Это три основные области, рассматриваемые в частях 2, 3 и 4 настоящей книги.

В следующих нескольких главах природа управляемых событиями программ для Windows исследуется в контексте создания минимальной работающей программы. Возможно, Вас удивит размер этой программы, но, тем не менее, она представляет собой минимум, обязательный для того, чтобы реагировать как на буферизуемые сообщения, так и на сообщения, которые "не ждут приглашения, чтобы войти". Минимальная программа для Windows создает единственное окно и демонстрирует, каким образом должна выполняться обработка поступающих в окно сообщений. Рассматривая каждый фрагмент программы, мы будем описывать, каким образом он соотносится с общей структурой Windows в целом.

Часть 2. Минимальная программа для Windows

Глава 2. Минимальная программа для Windows

В данной главе представлена минимальная программа для Windows MIN.EXE. Программу MIN можно рассматривать как шаблон для создания всех прочих приводимых в книге программ. Окно, создаваемое программой MIN, можно перемещать по экрану, можно изменить его размер, закрыть его, минимизировать (уменьшить до размеров пиктограммы) или максимизировать (увеличить до размеров всего экрана). Другими словами, опытный пользователь Windows имеет возможность выполнять в отношении создаваемого этой программой окна все стандартные разрешенные действия и операции.

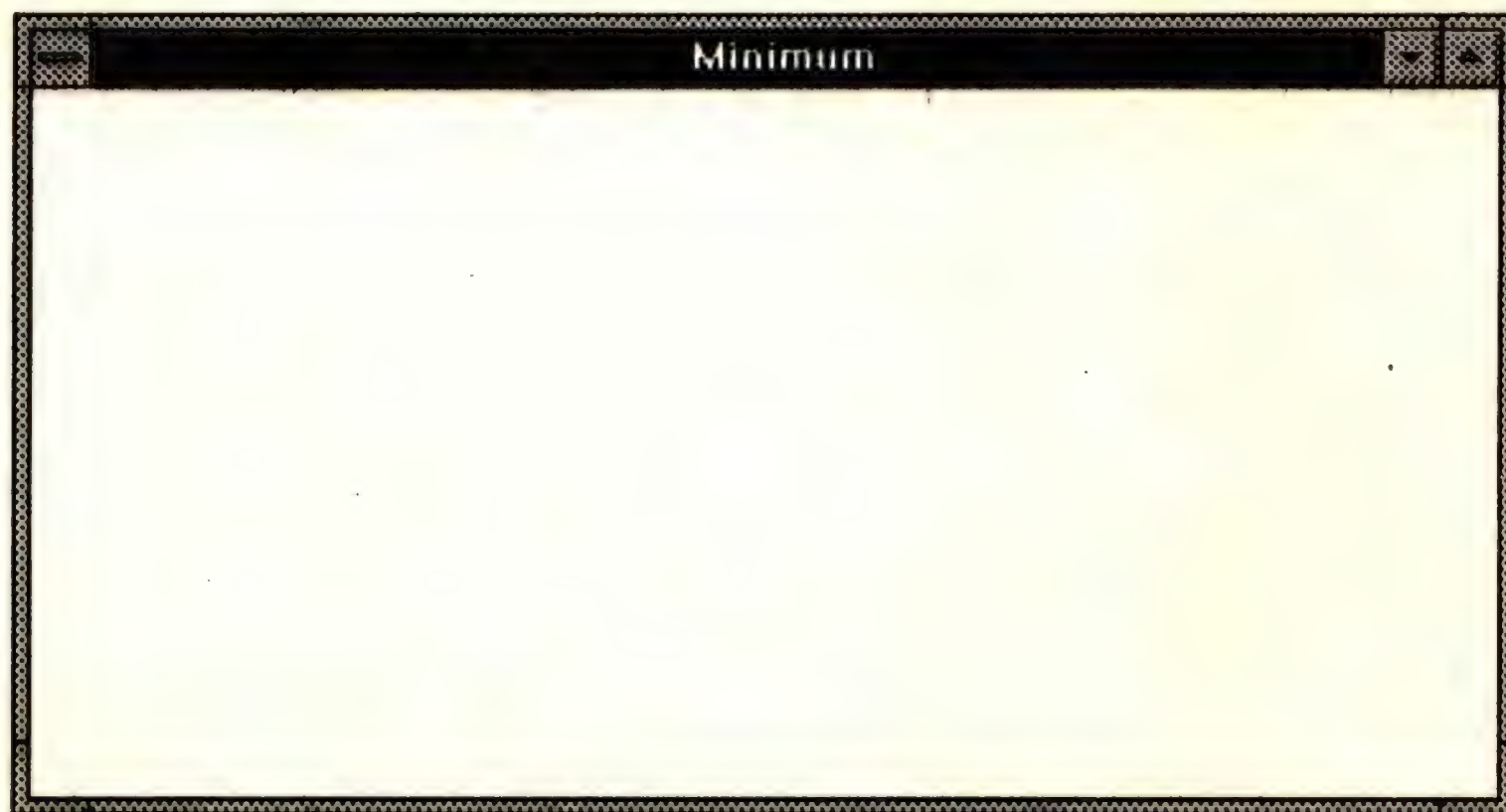


Рис.2.1. Окно, создаваемое минимальной программой Windows.

Таблица 2.1. Файлы, составляющие минимальную программу для Windows.

MAKEFILE.MAK	Файл для автоматизации процесса построения программы
MIN.LNK	Файл с командной строкой компоновщика
MIN.CPP	Исходный код на языке C++
MIN.RC	Файл ресурсов для объектов данных пользовательского интерфейса
MIN.DEF	Файл определения модуля, используемый компоновщиком
MIN.ICO	Файл пиктограмм, относится к ресурсам
MIN.CUR	Файл курсоров, также относится к ресурсам

В табл.2.1 перечислено семь файлов, составляющих нашу минимальную программу для Windows. Пять из них являются текстовыми (MAKEFILE.MAK, MIN.LNK, MIN.CPP, MIN.RC и MIN.DEF) и приводятся в сопровождающих данное обсуждение листингах. Остальные два файла содержат графические образы (MIN.CUR и MIN.ICO). Содержимое этих файлов в графическом представлении можно посмотреть в утилите Resource Workshop фирмы Borland. Файл MIN.CUR определяет курсор мыши в виде руки. Файл MIN.ICO содержит пиктограмму, которая появляется на экране, когда окно программы минимизировано.



Рис.2.2. Пиктограмма и курсор, используемые минимальной программой Windows.

MAKEFILE.MAK

.AUTODEPEND

```
# Определения транслятора
INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;C:\BORLANDC\INCLUDE
CC = bcc -c -D_CLASSDLL -H -ml -WS -w -IS(INC)

# Неявные правила
.c.obj:
$(CC) $(S< )

.cpp.obj:
$(CC) $(S< )

# Явные правила
Min.exe: Min.res Min.def Min.obj
tlink /c/C/n/P-/Twe/x @Min.LNK
RC Min.res Min.exe

# Зависимости отдельных файлов
Min.obj: Min.cpp

Min.res: Min.rc Min.cur Min.ico
RC -R -FO Min.res Min.RC
```

MIN.LNK

```
c:\borlandc\lib\c0wl.obj+
min.obj
min,min
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwl.lib+
import.lib+
mathl.lib+
cl.lib
min.def
```

MIN.CPP

```
/*
MIN.CPP Минимальная программа C++. Высвечивает окно, которое может
быть перемещено, изменено в размерах, минимизировано или максимизировано
*/
#include <owl.h>
```



```

/*
Определения классов
*/
class TMinApplication : public TApplication
{
public:
    TMinApplication (LPSTR lpszName, HANDLE hInstance,
                     HANDLE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow);
    virtual void initMainWindow ();
};

class TMinWindow : public TWindow
{
public:
    TMinWindow (PTWindowsObject pwParent, LPSTR lpszTitle,
                PTModule pmModule);
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WNDCLASS&);
};

/*
Основная функция: WinMain
*/
int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
{
    TMinApplication Min ("MIN", hInstance, hPrevInstance,
                          lpszCmdLine, nCmdShow);
    Min.Run();
    return Min.Status;
}

/*
Компонентная функция класса прикладной программы
*/
TMinApplication::TMinApplication (LPSTR lpszName, HANDLE hInstance,
                                   HANDLE hPrevInstance, LPSTR lpszCmdLine,
                                   int nCmdShow)
    :TApplication (lpszName, hInstance, hPrevInstance,
                  lpszCmdLine, nCmdShow)
{
    /* Инициализация прикладной программы */
}

/*
Компонентная функция класса прикладной программы
*/
void TMinApplication::InitMainWindow ()
{
    MainWindow = new TMinWindow (NULL, "Minimum", NULL);
}

/*
Компонентная функция класса TMinWindow
*/
TMinWindow::TMinWindow (PTWindowsObject pwParent,
                         LPSTR lpszTitle, PTModule pmModule)
    :TWindow (pwParent, lpszTitle, pmModule)

```



```

{
/* Здесь инициализируется окно */
}

/*
Компонентная функция класса TMinWindow
*/
LPSTR TMinWindow::GetClassName ()
{
return "MIN:MAIN";
}

/*
Компонентная функция класса TMinWindow
*/
void TMinWindow::GetWindowClass (WNDCLASS& wc)
{
TWindow::GetWindowClass (wc);
wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
wc.hCursor=LoadCursor (wc.hInstance, "hand");
}

```

MIN.RC

```

snapshot icon min.ico
hand cursor min.cur

```

MIN.DEF

NAME MIN

EXETYPE WINDOWS

DESCRIPTION 'Min -- Minimum Windows Program'

CODE MOVEABLE DISCARDABLE
DATA MOVEABLE MULTIPLE

HEAPSIZE 512
STACKSIZE 5000

При близком рассмотрении Вы обнаружите, что все эти исходные файлы используют компилятор командной строки Borland - BCC.EXE. Фирма Borland также предлагает интегрированную среду разработки, которая называется Programmer's Platform и позволяет редактировать, компилировать и компоновать исходные файлы. К сожалению, создаваемые этой программой файлы проекта являются двоичными и напечатать их в данной книге не представилось возможным. Подробное описание установок Programmer's Platform для построения программ для Windows см. в приложении G.

Имея размер всего лишь в 80 строк, минимальная программа может оказаться самой большой из всех, с которыми Вам когда-либо приходилось работать. Такой размер отражает объем работы, которую программа автоматически выполняет для подключения к библиотекам Windows 3 и библиотекам объектов классов OWL. Однако после установления программой всех необходимых связей Windows и OWL совместно делают за Вас большую часть всей работы.

Например, Windows обеспечивает по умолчанию поддержку различных оконных элементов. Рассмотрим системное меню с его семью командами: Restore (восстановить), Move (переместить), Size (изменить размер), Minimize (минимизировать до размеров пиктограммы), Maximize (максимизировать до размеров окна), Close (закрыть) и Switch To (переключиться на). Система Windows сама автоматически поддерживает работоспособность всех этих команд, требуя от Вашей программы лишь минимум действий. В свою очередь, библиотеки OWL структурированы таким образом, чтобы сообщения, генерируемые при выборе элементов этого меню, доставлялись в нужные точки и обработка их происходила правильно.

Каждая часть минимальной программы является обязательной для обеспечения правильного взаимодействия объектов OWL с Windows. Структура данной программы позволяет осуществлять ее расширение, так что на базе MIN мы сможем создавать более сложные программы. К тому же она представляет собой пример такой структуры, которую должны иметь любые программы для Windows, создаваемые с использованием библиотек OWL.

Возможно, Вам не терпится скомпилировать и запустить эту программу, поэтому прежде всего рассмотрим средства, которые понадобятся для построения программ под Windows. Вы уже знакомы с некоторыми из этих средств, такими, как компилятор и компоновщик Borland C++. Обратите внимание на ключи компилятора, которые должны быть установлены для генерации им Windows-совместимого кода. Что же касается компоновщика, то ему нужен специальный входной файл - файл определения модуля (с расширением .def). Прочие описываемые средства входят в число специфических средств среды разработки программ для Windows, например, Resource Compiler и Resource Workshop.

Описав инструментальные средства разработки, мы сделаем небольшой экскурс в следующую главу, чтобы поговорить о некоторых соглашениях, принимаемых при написании программ для Windows. Сюда входят Соглашение о Венгерской записи Имен, которое используется при записи имен структур данных и переменных. Мы также рассмотрим важные включаемые файлы, которые понадобятся Вам, чтобы иметь возможность воспользоваться библиотеками Windows и OWL, а именно WINDOWS.H и OWL.H. Эти файлы содержат обязательные символические константы, структуры данных, прототипы функций и определения классов C++.

В последующих главах мы рассмотрим код нашей минимальной программы для Windows. Мы начнем с объекта прикладной программы, определяемого классом TApplication. Экземпляр этого класса создается в WinMain, в функции, являющейся точкой входа в любую программу для Windows. Обычно WinMain создает объект прикладной программы (или объект класса "прикладная программа"), после чего входит в (почти) бесконечный цикл, принимая в нем аппаратно-генерируемые сообщения. Сообщение представляет собой единицу ввода в программу для Windows, и составляет тот "квант времени", который используется при планировании работы многозадачной (невывесняющей) системы Windows.

В главе 5 мы рассмотрим классы TWindow и TWindowObject, являющиеся предками оконного объекта (или объекта класса "окно") в программе MIN. Может показаться, что три компонентные функции оконного объекта программы MIN не столь важны, однако они играют специальную роль в каждой программе для Windows. Мы покажем, как программа MIN использует классы TWindow и TWindowObject для подключения окна к библиотекам Windows. В частности, рассмотрим оконную процедуру по умолчанию. Это библиотечная подпрограмма Windows, обеспечивающая минимальный набор операций для функционирования окна. Мы подробно остановимся также на компонентных функциях, которые Вы, вероятнее всего, захотите переопределить для создания собственных интересных и полезных программ для Windows.

Завершится данный раздел книги детальным описанием различных типов сообщений, с которыми Вы встретитесь в качестве программиста, пишущего под Windows. Вы увидите, что в WINDOWS.H определено около 250 различных сообщений. В разделе "Таксономия сообщений" мы выделим восемь их категорий.

Начнем с изучения различных инструментальных средств, используемых при построении нашей минимальной программы для Windows.

Механизм компиляции и компоновки MIN.EXE

При написании программ под Windows Вы будете пользоваться средствами разработки, которые знакомы Вам по работе в других средах разработки - это компилятор языка программирования и компоновщик. Другими инструментами, специфичными для среды разработки Windows, являются компилятор ресурсов Resource Compiler и Resource Workshop. Эти средства используются при создании объектов пользовательского интерфейса и слиянии их в исполняемый файл программы. Практически любая создаваемая Вами программа для Windows потребует использования тех или иных ресурсов. Например, главы 11 и 14 познакомят Вас с использованием ресурсов для создания меню и блоков диалога. Программа MIN содержит всего два ресурса: курсор и пиктограмму.

ПРИМЕЧАНИЕ для программистов, работавших с Apple Macintosh. Ресурсы Windows похожи на ресурсы Macintosh. Однако, в отличие от программы для Macintosh, программа под Windows не имеет возможности изменять образ ресурса, хранимый на диске. Другими словами, ресурсы Windows содержат данные, доступные только для чтения.

Если Вы используете интегрированную среду разработки Borland Programmer's Platform, Вам не нужно заботиться о правильной установке ключей компилятора и компоновщика. Посмотрите в приложении G, как сделать соответствующие установки в самой интегрированной среде. Если же Вы, подобно нам, привержены "старомодному" способу построения программ с помощью командной строки, читайте дальше. Прежде всего мы рассмотрим утилиту MAKE, которая координирует работу компилятора C++, компоновщика и компилятора ресурсов.

Утилита MAKE

Утилита MAKE фирмы Borland происходит от утилиты с тем же именем из Unix. В целом утилиты MAKE автоматизируют процесс создания программы для минимизации избыточных операций, MAKE компилирует и компоует только файлы, измененные с момента последней компиляции или компоновки. Для столь интеллектуального поведения MAKE использует входной файл (MAKE-файл), описывающий взаимосвязь каждого файла программы с файлом, получаемым на выходе. Утилита MAKE фирмы Borland также имеет возможность определять взаимные зависимости между файлами исходных кодов (.CPP-файлами) и включаемыми файлами (.H-файлами) на базе информации в скомпилированных объектных файлах (.OBJ-файлах). Ниже показан используемый для построения MIN-файл.

MAKEFILE.MAK:

.AUTODEPEND

Определения для транслятора

INC=C:\BORLAND\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;C:\BORLANDC\INCLUDE

CC = bcc -c -D_CLASSDLL -H -ml -WS -w -I\$(INC)

Неявные правила

.c obj:

\$(CC) \$(<)

.cpp.obj:

\$(CC) \$(<)

Явные правила


```
Min.exe: Min.res Min.def Min.obj
tlink /c/C/n/P-/Twe/x @Min.LNK
RC Min.res Min.exe

# Зависимости конкретных файлов
Min.obj: Min.cpp

Min.res: Min.rc Min.cur Min.ico
RC -R -FO Min.res Min.RC
```

В этом файле есть ссылка на файл командной строки компоновщика, MIN.LNK, показанный ниже:

```
c:\borlandc\lib\cOwl.obj+
min.obj
min,min
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwl.lib+
import.lib+
mathl.lib+
cl.lib
min.def
```

Оператор AUTODEPEND в MAKE-файле заставляет утилиту MAKE прочесть информацию о зависимостях, помещаемую компилятором в .OBJ-файлы. Это гарантирует перекомпиляцию программных файлов в случае изменения любого включаемого (.H) файла, на который в программе есть ссылка. Это помогает обеспечить постоянную актуальность данных в MAKE-файле.

Вы могли заметить, что некоторые символы в MAKE-файле имеют специальный смысл. В частности, знак фунта (#) обозначает начало строки комментария, который продолжается до конца строки. Обратная косая (\) указывает на продолжение строки (кроме строки комментария).

В целом утилита MAKE определяет правила, которым необходимо следовать в своей работе из чтения MAKE-файла. Иногда это могут быть "неявные правила", как, например:

```
.cpp.obj:
$(CC) {$< }
```

Этот оператор говорит о том, что файлы с расширениями .OBJ зависят от файлов с расширениями .CPP при условии равенства их имени. Выполняемая команда - это команда, хранимая в переменной CC, определяемой следующим образом:

```
CC = bcc -c -D_CLASSDLL -H -ml -WS -w -I$(INC)
```

Неявное правило будет вызвано, когда в MAKE-файле встретится, например, следующий оператор:

```
Min.obj: Min.cpp
```

Неявные правила полезны, например, если у Вас имеется множество исходных файлов на языке C++, скомпилированных с помощью одинаковой командной строки (строк). Иногда же имеет смысл воспользоваться "явными правилами". Их используют, если в MAKE-файле такая зависимость встречается всего один раз, например, это может быть вызов компоновщика. Ниже показано явное правило из MAKE-файла:

```
Min.exe: Min.res Min.def Min.obj
tlink /c/C/n/P-/Twe/x @Min.LNK
RC Min.res Min.exe
```


В случае изменения в одном из трех входящих в зависимость файлов (MIN.RES, MIN.DEF или MIN.OBJ) программа MIN.EXE будет обновлена посредством выполнения команд второй и третьей строк данного правила. Эти команды запускают компоновщик (TLINK.EXE) и компилятор ресурсов (RC.EXE).

Ключи компилятора

Рассмотрим поближе передаваемые компилятору ключи. Имя используемого нами компилятора командной строки - BCC.EXE. Этот компилятор использует Интерфейс Защищенного Режима DOS (DOS Protected Mode Interface - DPMI). DPMI дает программам доступ к расширенной памяти (память за пределами 1-мегабайтной границы адресного пространства процессоров 80286 и старше). Рассмотрим ключи компилятора по отдельности.

Ключ -с говорит компилятору, что нужна компиляция без компоновки. Основное его достоинство состоит в том, что он позволяет создавать программу в виде нескольких исходных файлов на языке C. Но он также необходим и потому, что программам для Windows требуются специальные команды для компоновщика.

Ключ -D определяет идентификатор для препроцессора компилятора. В данном случае -D_CLASSDLL создает идентификатор _CLASSDLL. Возможно, Вам известно, что определение такого рода идентификаторов позволяет написать код для условной компиляции. Например, блок кода между операторами #if и #endif будет включен в компиляцию, только если идентификатор _CLASSDLL определен:

```
#if defined (_CLASSDLL)
x = 15;
TextOut(hdc,x,y,"Вывести только если _CLASSDLL определен", 36);
#endif
```

Оператор _CLASSDLL, на который ссылается главный включаемый файл библиотеки OWL, OWL.H, создает программу для Windows, использующую динамически компонуемую библиотеку OWL.DLL. (Подробное сравнение динамической и статической компоновки см. в главе 19.)

Ключ -H заставит компилятор использовать прекомпилированные файлы заголовков (.H-файлы). Эта возможность помогает ускорить компиляцию, в частности, при работе с большими включаемыми файлами. Директива компилятора:

```
#pragma hrdstop
```

позволяет провести линию между неизменяющимися и обычно изменяющимися файлами заголовка. Все файлы заголовка до этого оператора будут включены в прекомпилированный заголовок, и идущие ниже этой директивы файлы будут компилироваться всякий раз при компиляции исходного файла.

Ключ -ml запрашивает модель большой памяти. Модель памяти описывает умолчания, используемые при адресации памяти. Модель большой памяти означает, что для ссылок как на код, так и на данные, будут по умолчанию генерироваться дальние (far) указатели. При создании программ для Windows, использующих динамически компокуемые библиотеки OWL (OWL.DLL, BCRTL.DLL и BWCC.DLL), Вы должны задавать модель большой памяти.

В документации ключ -WS описан как ключ, запрашивающий "интеллектуальный экспорт". Это способ, с помощью которого дальние (far) функции устанавливают регистр сегмента данных (DS - data symbol) для Вашего кода. Во многих случаях библиотекам Windows необходимо обратиться к Вашим подпрограммам. Например, для поддержки окон и блоков диалога библиотекам Windows передается адрес специальной функции "обратного вызова". Если библиотекам Windows необходимо связаться с Вашими окнами или блоками диалога, происходит обращение к функции обратного вызова. Для того чтобы "перекинуть мост" между Вашими кодом и данными, необходим соответствующий механизм, в этом случае механизм интеллектуального экс-

порта. Подробнее мы рассмотрим его в главе 19 при описании механизма динамической компоновки Windows.

Ключ -w разрешает выдачу всех предупреждений. Это позволяет сделать Вашу программу более защищенной от возможных ошибок. Некоторые предупреждения являются разрешенными по умолчанию. Данный ключ распространяет это разрешение на все остальные предупреждения.

Ключ -I определяет маршрут поиска включаемых (.H) файлов. Разумеется, установки, выполняемые для этого ключа, будут зависеть от того, где Вы решили установить компилятор и файлы библиотек OWL. В нашем случае мы воспользовались инсталляционными маршрутами по умолчанию как для компилятора, так и для файлов библиотек классов OWL.

Файл ресурсов

Файл MIN.RC содержит список ресурсов, которые должны быть объединены с MIN.EXE. С точки зрения Менеджера Памяти Windows ресурс представляет собой объект данных со статусом "только чтение". Отделяя данные, доступные только для чтения, от доступных для чтения и записи, программа помогает Менеджеру Памяти Windows оптимизировать использование памяти. Почти все объекты пользовательского интерфейса хранятся в виде ресурсов, каждая программа для Windows (осознанно или неосознанно) помогает оптимизировать использование памяти. В табл.2.2 перечислено восемь типов предопределенных ресурсов. Всем этим объектам соответствуют функции Windows, предназначенные для их создания и управления ими. Если Вы используете большие блоки данных с доступом только для чтения, то можете создавать собственные пользовательские типы ресурсов, чтобы еще полнее оптимизировать использование памяти.

Ресурсы обычно считываются в память по мере необходимости, хотя Вы можете объявить ресурс как PRELOAD, т. е. загружаемым в память при начальной загрузке программы. Будучи считанным в память, ресурс обычно располагается в блоке памяти, объявленном как DISCARDABLE, т.е. рабочий, временный. Такая память может быть очищена и освобождена Менеджером Памяти Windows по мере надобности.

Таблица 2.2. Предопределенные ресурсы Windows.

Тип Ресурса	Подробно рассматривается
Таблица кодовых комбинаций клавиш	Глава 11
Битовые образы или растры Курсоры	— " — 16
Шаблон блока диалога	— " — 14
Шрифты	— " — 10
Шаблон меню	— " — 11
Строковая таблица	— " — 18

Наша минимальная программа для Windows использует два типа ресурсов: пиктограмму и курсор. Для создания собственных пиктограмм и курсоров используйте Whitewater Resource Toolkit из Borland Resource Workshop. Мы использовали Resource Workshop.

Курсоры - это битовые образы, реагирующие на перемещение мыши. Они позволяют пользователю "указывать" на различные объекты на экране дисплея. Для нашей минимальной Программы для Windows мы создали курсор в виде руки с указывающим пальцем. Для создания этой картинки мы воспользовались программой Paintbrush, поставляемой вместе с Windows, после чего перенесли этот образ в Resource Workshop через Clipboard.

При создании пиктограммы MIN мы приняли несколько иной подход. Поскольку пиктограмма призвана напоминать пользователю о существовании такой программы, мы создали пиктограмму из "моментального снимка" MIN, сделанного вс время выполнения этой программы при помощи встроенного средства Windows. При нажатии клавиши [PrtScr] ("Распечатать содержимое экрана") Windows помещает моментальный снимок всего экрана в Clipboard. Если же нажать [Alt]+[PrtScr], то в Clipboard попадет только образ текущего активного окна.

Мы воспользовались комбинацией [Alt]+[PrtScr], чтобы сделать моментальный снимок окна MIN.EXE. Затем мы вставили его из Clipboard в рабочую область Resource Workshop. Немножко "почистив" картинку, мы получили готовую пиктограмму. Далее мы поместили пиктограмму и курсор в соответствующие файлы и задали в файле ресурсов MIN.RC для каждого из них имя ресурса, тип ресурса и имя файла, где этот ресурс определен:

```
snapshot icon min.ico  
hand cursor min.cur
```

При построении программы компилятор ресурсов копирует каждый заданный там ресурс в MIN.EXE. Это делается для того, чтобы каждая программа для Windows представляла собой автономный исполняемый файл, минимально зависящий от внешних файлов.

ЗАМЕЧАНИЕ. Средство "перехвата" ("моментального снимка") экрана исключительно полезно при создании документации по программам для Windows. Этим методом мы "сфотографировали" все необходимые экраны для данной книги. Был выбран монохромный драйвер экрана VGA и лазерный принтер HP LaserJet II. Снимки экрана были переданы в редактор Microsoft Word for Windows, который позволил вывести эти снимки на печать.

Компоновщик

Работа компоновщика состоит в построении исполняемого (.EXE) файла программы из объектных (.OBJ) и библиотечных (.LIB) файлов. Используемый нами компоновщик командной строки TLINK.EXE создан фирмой Borland. Он вызывается из MAKEFILE.MAK с помощью следующей командной строки:

```
tlink /c/C/n/P-/Twe/x @Min.LNK
```

Ключ /с устанавливает значимость регистра букв в процессе компоновки. Обычно компоновщик Borland не обращает внимания на регистр букв, поскольку основной продукт Borland - компилятор Паскаля тоже не учитывает регистр. Данный ключ решает эту проблему, переключая режим учета регистра на принятый в стандарте C/C++.

Ключ /C заставляет компоновщик рассматривать импортируемые и экспортируемые идентификаторы с учетом регистра. Этот ключ необходим, поскольку программы на C++ генерируют идентификаторы с буквами как верхнего, так и нижнего регистра, в то время как компоновщик обычно не учитывает регистр в таких именах.

Ключ /n заставляет компоновщик игнорировать библиотеки по умолчанию. Мы сами зададим компоновщику нужные имена библиотек.

Ключ /P- отменяет упаковку программных сегментов. По умолчанию компоновщик объединяет программные сегменты вместе. Однако, как описано в главе 18, Вам понадобится самостоятельно управлять сегментацией программы, поэтому мы рекомендуем Вам использовать этот ключ.

Ключ /Twe заставит компоновщик создать исполняемый (.EXE) файл для Windows. Этот ключ необходим, так как компоновщик умеет еще и создавать файлы динамически компокуемых библиотек (.DLL) для Windows и исполняемые (.EXE) файлы для DOS.

Ключ /x отменяет создание компоновщиком файла карты распределения памяти. Файл карты памяти содержит все идентификаторы программы и может использоваться как справочный материал о том, как именно программа фактически использует память. Пока мы с помощью этого ключа сэкономим место на диске, отменив создание этого файла.

Файл командной строки компоновщика MIN.LNK ссылается на следующие файлы:

C0WL.OBJ	Содержит код начальной загрузки и код выхода из программы для Windows с моделью большой памяти. Выполняет некоторые действия по инициализации, после чего вызывает WinMain. После завершения последней перед выходом в систему код из этого объекта выполняет некоторые завершающие действия по очистке памяти.
OWL.LIB	Сообщает компоновщику о функциях в OWL.DLL динамически компоновываемой библиотеке объектов. Этот файл обязан присутствовать при работе программы, маршрут к нему задается в переменной PATH.
CRTLL.LIB	Сообщает компоновщику о функциях в BCRTL.DLL, части библиотеки исполняющей системы Borland C, перенесенной в динамически компоновываемую библиотеку Windows.
CWL.LIB	Содержит статически компоновываемые функции из библиотеки исполняющей системы Borland C. (Ссылка на этот файл должна стоять выше ссылки на CL.LIB.)
IMPORT.LIB	Сообщает компоновщику о функциях в динамически компоновываемых библиотеках Windows 3.0 (KERNEL.EXE, USER.EXE и GDI.EXE).
MATH.LIB	Содержит версию библиотеки математических функций Borland для модели большой памяти.
CL.LIB	Представляет собой стандартную библиотеку исполняющей системы Borland C для модели большой памяти (для функций, не переопределенных в CRTLL.LIB или CWL.LIB).

Компоновщик и файл определения модуля

Помимо .OBJ- и .LIB-файлов, при создании программ под Windows компоновщик принимает на входе файл определения модуля (.DEF-файл). Роль этого файла состоит в описании структуры и организации программы. .DEF-файл можно рассматривать как набор ключей компоновщика. Остановимся поочередно на каждом из операторов файла определения модуля MIN.DEF.

NAME MIN (ИМЯ)

В Windows весь код и все данные организованы в виде модулей. Существует два типа модулей: исполняемые программы и библиотеки динамической компоновки. Ключевое слово NAME (ИМЯ) определяет модуль как исполняемую программу. (Для того чтобы определить библиотеку динамической компоновки, нужно воспользоваться альтернативным ключевым словом LIBRARY - БИБЛИОТЕКА). Имя модуля помещается после ключевого слова NAME (или LIBRARY). Имя модуля нашей программы - MIN. Хотя ранние версии Windows использовали для различения модулей имена этих модулей, в Windows 3.0 при различении модулей высший приоритет имеют имена файлов.

Следующий оператор заставляет компоновщик создать программу для Windows:

EXETYPE WINDOWS (ТИП ИСПОЛНЯЕМОГО ФАЙЛА)

Это необязательный оператор, поскольку ключ /Twe уже сообщил компоновщику, что должна создаваться программа для Windows. Тем не менее, мы лучше будем использовать этот оператор, так как следующие версии компоновщика смогут создавать программы для OS/2. Тогда при необходимости мы сможем заменить WINDOWS на OS2.

Оператор **DESCRIPTION** (ОПИСАНИЕ) вставляет в исполняемый файл некоторый текст:

DESCRIPTION 'Min -- Минимальная Программа для Windows.

Обычно этот оператор используется для того, чтобы задать информацию об авторских правах на программу или ее версии. Чтобы посмотреть описание программы (или динамически компокуемой библиотеки) для Windows, запустите утилиту **EXEHDR.EXE**.

Оператор **CODE** устанавливает статус для сегментов кода программы для Windows:

CODE MOVEABLE DISCARDABLE (КОД ПЕРЕМЕСТИМЫЙ УНИЧТОЖИМЫЙ)

В главе 18 мы увидим, как использование оператора **SEGMENTS** позволяет задать статус для отдельных программных сегментов. Однако пока для нас достаточно будет оператора **CODE**. Объявление слова **MOVEABLE** (ПЕРЕМЕЩАЕМЫЙ) (его альтернатива **FIXED** - ФИКСИРОВАННЫЙ) дает возможность перемещать сегмент в памяти. Объявление **DISCARDABLE** (УНИЧТОЖИМЫЙ) означает, что при недостатке памяти программный сегмент может быть удален из оперативной памяти системы. Механизм динамической компоновки позволяет перемещать код в памяти, равно как и удалять из памяти таким образом, что он будет полностью прозрачным для Вашей программы.

Оператор **DATA** аналогичен оператору **CODE**, за исключением того, что он устанавливает диспозицию памяти для сегмента данных программы:

DATA MOVEABLE MULTIPLE (ДААННЫЕ ПЕРЕМЕСТИМЫЕ МНОЖЕСТВЕННЫЕ)

С помощью **MOVEABLE** выполняется перемещение сегмента данных в памяти. (Когда программе доставляется сообщение, сегмент данных фиксируется в памяти, чтобы избежать неожиданных результатов. Это может вызвать проблемы для Windows в реальном режиме, но поскольку компилятор Borland создает прикладные программы лишь для защищенного режима, то Вас это не должно волновать.) Объявление **MULTIPLE** является в программах для Windows стандартным. Оно обеспечивает одновременное выполнение нескольких копий одной и той же программы.

Оператор **STACKSIZE** устанавливает размер стека программы:

STACKSIZE 5000 (РАЗМЕР СТЕКА)

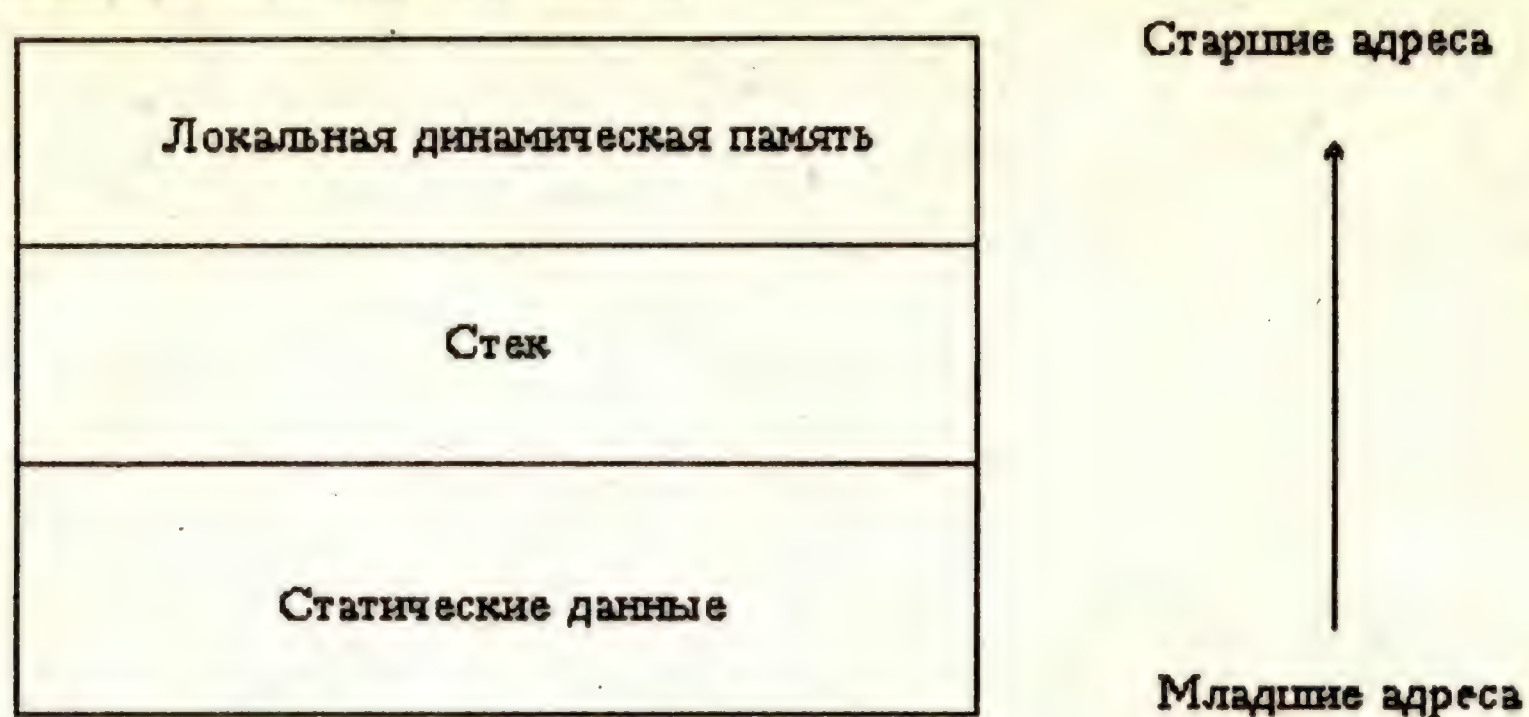
Стек используется для выполнения трех задач: для хранения локальных переменных, передачи параметров в вызываемые функции и сохранения адреса возврата для "анонимных вызовов". Программы для Windows должны иметь стек не менее 5 Кбайт. (Если задано меньшее значение, то загрузчик Windows сам автоматически выделит стек размером 5 Кбайт.) 2 Кбайта из общего размера стека предназначаются для использования Вашей программой. Остальные 3 Кбайта использует Windows. Когда программа вызывает библиотечную функцию Windows, аргументы в нее передаются через стек программы. Для программ с большим числом локальных переменных или для рекурсивных операций следует предусматривать больший размер стека.

Оператор **HEAPSIZE** устанавливает исходный размер локальной динамической памяти программы. В нашей минимальной программе задано:

HEAPSIZE 512 (РАЗМЕР КУЧИ)

Локальная динамическая память - это одна из двух областей, в которых возможно динамическое выделение памяти. (Второй областью является глобальная динамическая память системы). Локальная динамическая память является частным ресурсом памяти программы и находится в сегменте данных программы. Ниже показана схема распределения памяти сегмента данных программы. Оператор **HEAPSIZE** устанавливает исходный размер локальной динамической памяти. Затем локальная ди-

намическая память может быть увеличена, и это увеличение ограничено лишь предельным размером сегмента - 64 Кбайта.



....
Рис.2.3. Схема распределения памяти сегмента данных программы.

Теперь Вы владеете достаточной информацией, чтобы ввести минимальную программу для Windows. Мы предполагаем, что сейчас Вы как раз это и сделаете. Она заодно позволит Вам проверить, правильно ли установлена среда разработки. Кроме того, Вы сможете ознакомиться с программой поближе, прежде чем мы углубимся "внутрь" этой программы.

В следующей главе мы рассмотрим некоторые соглашения, которые повсеместно приняты в отношении программ для Windows, а также соглашения, действующие для библиотек классов OWL.

Глава 3. Соглашения о программировании для Windows и OWL

Среди всех аспектов работы программиста создание имен переменных, скорее всего, окажется в самом конце списка. Однако создание имен переменных - это задача, которую приходится выполнять каждому программисту. Мы хотим познакомить Вас с Венгерской Записью Имен, принятой в практике программирования для Windows. Вас, скорее всего, смутит тот факт, что с Венгерской Записью Вы встретились впервые. Однако со временем мы убедились в исключительной пользе такого способа записи и попытаемся представить его Вам.

После Венгерской Записи мы поговорим о различных включаемых файлах, которые понадобятся Вашей программе. Вы уже, наверное, обратили внимание на то, что одной из первых строк в нашей программе для Windows стояла строка:

```
#include <owl.h>
```

Этот включаемый файл подключает к программе большую часть из необходимых ей библиотечных файлов OWL, а также WINDOWS.H.

Файл WINDOWS.H представляет собой очень большой (120 Кбайт) файл данных, описывающий элементы, необходимые для подключения к Windows. Вы можете распечатать его и ознакомиться с содержимым. Вы увидите в нем "пустые" определения API в системе Windows в том виде, в котором они становятся доступными Вашей программе.

Затем мы рассмотрим "логические номера", представляющие собой широко распространенный в Windows тип данных. Логические номера еще называют на программистском жаргоне "магическими номерами" или "волшебными пирожками". Какой бы термин Вы не предпочли, логические номера идентифицируют объекты, создаваемые одними или другими частями системы. Смысл логического номера известен только той части системы, которая создала объект и назначила данный логический номер.

И, наконец, мы коснемся вопроса, связанного со старыми программами для Windows: приведения типов. В прежних версиях Windows приведение типов было неизбежной операцией, однако с развитием компиляторов оно превратилось в устаревшую практику. Ниже мы объясним, что использование приведений типов может оказаться опасным из-за возможности внедрения в программу ошибок, которые чрезвычайно трудно локализовать.

Начнем с того, что рассмотрим венгерскую запись имен.

Венгерская запись имен

Запись имен - это соглашение о правилах создания имен переменных и функций. Она широко распространена среди программистов, работающих в Windows, так как облегчает чтение и сопровождение программ. Венгерская запись получила свое название от национальности человека, впервые применившего этот способ записи, Чарльз Симони. Название "венгерская запись" хорошо подходит к данному соглашению, так как для большинства программистов этот способ записи имен поначалу непонятен, как венгерский язык. Симони разработал свой способ записи имен и обосновал в докторской диссертации, посвященной программированию. Венгерская запись широко распространилась в Xerox PARC, где Симони одно время работал. Затем запись стали использовать в Microsoft, куда перешел Симони, и практически она стала стандартом для всех работающих в Windows.

Создать полезное имя не так просто, как кажется. Должны ли имена быть краткими? Конечно, короткое имя легче записать, однако создаваемый таким образом код будет трудно понимаем. Вы знаете, о чем идет речь, если когда-нибудь читали программы на Бейсике, где приняты очень короткие имена. Что можно сказать о переменных A и B в следующем фрагменте программы на Бейсике?


```

20 LET B=10
30 FOR A=1 TO 10
40 LET B=B+A
50 NEXT A

```

Если короткие имена неинформативны, то нельзя ли получать полезные имена, делая их длинными и описательными? Есть ли какие-то сомнения относительно того, что делает переменная `loopindex`? Однако такие имена могут оказаться слишком длинными и громоздкими. Рассмотрим список имен:

```

countofcharacters
numberoffiles
temporaryfilename
windowhandle
pointertoarrayofcharacters

```

Может быть, программу с такими именами и легче читать, но при этом их трудно набирать. Длинные имена также увеличивают вероятность опечатки.

Венгерская запись занимает среднее положение между этими двумя крайними подходами. В ней имена переменных создаются из короткого префикса и более длинного содержательного имени. Префикс описывает тип данных, к которому принадлежит переменная. В некоторых случаях префикс описывает и способ ее использования. Имя переменной иногда может состоять и только из одного префикса.

Вот несколько примеров венгерской записи:

```

char      ch;
char      achFile[128];
char far * lpszName;
int       cbName;

```

Отметим, что префиксы набраны строчными буквами, а длинные имена состоят как из строчных, так и из заглавных букв. Первая переменная `ch`, является префиксом для символьных данных. Это пример префикса, используемого в качестве имени переменной. Префикс в `achFile` состоит из двух частей: "a" означает, что это массив (от слова "array"), тогда как "ch" говорит о типе данных в массиве (от слова "character"). Префикс в `lpszName` также состоит из двух частей: "lp" означает "длинный указатель" (от "long pointer"), "sz" описывает тип данных, на которые указывает указатель, в данном случае это завершающаяся нулевым символом строка. Переменная `cbName` имеет префикс "cb", происходящий от слов "count of bytes" ("счетчик байтов").

Понимание венгерской записи помогает читать строки программы:

```

LPSTR      lpszName;
LPSTR      lpsz;
int        cbName;

for (lpsz = lpszName, cbName=0;
     lpsz != '\0';
     lpsz++, cbName++);

```

Префикс "cb" сообщает нам о том, что переменная `cbName` обязана содержать счетчик байтов. Поскольку так оно и есть, то становится совершенно понятным, что назначением приведенного фрагмента является определение числа символов в заканчивающейся пустым (Null) символом строке, к которой обращает Вас указатель `lpszName`. Имена, состоящие только из префикса, принято использовать для временных, вспомогательных переменных, как показано на примере переменной `lpsz`.

Помимо того, что венгерская запись улучшает читаемость кода, она еще позволяет избежать ряда досадных (но от того не менее частых) ошибок программирования. Освоившись с венгерской записью, Вы с первого взгляда будете видеть, что


```
lpszName = achFile;
```

является допустимым оператором, а следующий оператор недопустим:

```
lpszName = cbName;
```

И действительно, операция присвоения указателю адреса массива имеет смысл, тогда как присвоение указателю значения счетчика байтов смысла не имеет.

Чтобы хорошо освоить венгерскую запись, необходимо выучить префиксы. Ниже приводится список широко распространенных в программировании для Windows префиксов.

Префикс	Тип данных
a	Массив (составной тип)
ch	Символ
cb	Счетчик байтов
dw	Длинное целое без знака (WINDOWS.H typedef: DWORD)
h	Логический номер -- 16-битовый идентификатор
hdc	Логический номер контекста устройства
hwnd	Логический номер окна
i	Индекс (составной тип)
l	Длинное целое (WINDOWS.H typedef: LONG)
lp	Дальний (или длинный) указатель (составной тип)
n	Целое
np	Ближний (или короткий) указатель (составной тип)
pt	Точка, описанная парой координат x,y (WINDOWS.H typedef: RECT)
sz	Заканчивающаяся нулевым символом строка
w	Целое без знака (WINDOWS.H typedef: WORD)

Отметим, что некоторые из этих префиксов являются префиксами составного типа. Это означает, что они могут являться префиксами к другим префиксам. Следующий фрагмент показывает, как два составных префикса, "a" и "i", определяют переменные, которые могут быть использованы совместно:

```
char ch;  
int ich; /* Индекс символьного массива */  
char achName[64]; /* Символьный массив */  
...  
ch = achName[ich];
```

Стоит отметить, что никакого "официального" списка префиксов не существует. Поскольку любой программный проект так или иначе имеет собственные уникальные типы данных, то неизбежно и создание префиксов, описывающих эти типы. В целом, однако, преимущество венгерской записи состоит в относительно небольшом числе типов и договоренности между участниками команды, работающей над проектом относительно значений этих типов.

Венгерская запись часто применяется для имен функций. Мы встретили несколько "диалектов" венгерской записи. Один из них при описании функции объединяет глагол и существительное. Например, существует три библиотечных подпрограммы Windows с именами CreateWindow ("СоздатьОкно"), DrawText ("НарисоватьТекст") и LoadIcon ("ЗагрузитьПиктограмму"). В библиотеках Windows Вы встретите и другой "диалект": некоторые имена подпрограмм состоят только из существительных,

например, `DialogBox` ("БлокДиалога"). Для подпрограмм, предназначенных для преобразования одного типа в другой, обычными являются такие имена, как `XtoY` ("изXвY"). Например, библиотечная подпрограмма `Windows DPtoLP` выполняет преобразование физических точек устройства в логические точки программы.

В программах, построенных на использовании библиотек `OWL`, встречается специальная подпрограмма, которая называется "обработчиком сообщений". Система `Windows` посылает в Вашу программу сообщения по мере возникновения представляющих интерес событий. Например, сообщение `WM_PAINT` говорит о необходимости перерисовать окно. Сообщение `WM_LBUTTONDOWN` - о том, что произошло нажатие левой кнопки мыши. Все сообщения поступают в оконную процедуру. Библиотека `OWL` посылает эти сообщения определенным Вами обработчикам сообщений. Имя обработчика сообщений является производным от имени сообщения. Например, компонентная функция `WMPaint` обрабатывает сообщение `WM_Paint`, а `WMLButtonDown` - сообщение `WM_LBttnDown`.

Если Вы хотите придерживаться венгерской записи, то сможете создавать программные коды, которые легче читать и сопровождать. Даже если Вы не примете венгерскую запись за основу в своей работе, знакомство с ней облегчит Вам чтение примеров в данной книге и в другой литературе. Вам также легче будет читать и осмысливать определения, которые находятся во включаемом файле `Windows, WINDOWS.H`.

Соглашения об именах `OWL`

Библиотеки `OWL` фирмы `Borland` в целом придерживаются венгерской записи Имен, однако с некоторыми вариациями, своим появлением обязанными в основном тому факту, что библиотеки были перенесены в `C++` из Паскаля, так что они до сих пор обладают некоторым "привкусом" Паскаля. И хотя эти вариации иногда приводят к некоторой путанице, мы находим их полезными для различения частей программы, написанных для `OWL`, от частей, написанных строго для `Windows`.

Библиотеки `OWL` используют в качестве префикса, обозначающего имя класса, заглавную букву "T" от слова "type" (тип). Вот несколько определяемых в `OWL` классов:

TApplication
TControl
TDialog
TWindow
TWindowsObject

Мы будем использовать это соглашение в наших примерах программ. Например, программа `MIN` содержит определения двух классов: `TMinApplication` и `TMinWindow`.

Компонентные функции классов `OWL` соблюдают соглашение о венгерской записи Имен, используя в именах заглавные и строчные буквы. Отметим при этом, что `OWL` отходит от чистой венгерской записи в том, что касается имен компонентных данных классов. В этой книге в именах компонентных данных классов мы следовали венгерской записи, поскольку это позволяет различать компонентные данные из `OWL` от наших собственных. Мы рекомендуем Вам поступать так же.

Логические номера

Логический номер является идентификатором. Он представляет собой 16-битовое целое без знака. Подобно тому, как `MS DOS` назначает логический номер открываемому файлу, так и `Windows` назначит логические номера для идентификации собственных объектов. Логический номер не имеет смысла вне контекста, в котором он был назначен. Например, Вы не можете преобразовать логический номер в указатель и использовать его каким-либо осмысленным образом. Логический номер возвращается несколькими библиотечными подпрограммами `Windows`. При неудачном завер-

шении такая подпрограмма возвращает "пустой логический номер" (т.е. handle == NULL).

ПРЕДУПРЕЖДЕНИЕ! Будьте внимательны - в Windows на недопустимый логический номер указывает значение NULL, но недопустимый логический номер файла равен -1. Это связано с тем, что логические номера файлов назначаются MS DOS, а не Windows.

Двумя наиболее важными типами логических номеров в Windows являются логический номер окна и логический номер контекста устройства. Логический номер окна идентифицирует окно. Каждое окно в системе имеет уникальный логический номер. Все подпрограммы, манипулирующие окнами, принимают логический номер окна в качестве параметра. Если Вы знаете логический номер окна, то можете перемещать это окно, менять его размеры, делать его невидимым и практически выполнять с ним любые операции, какие только могут понадобиться.

Логические номера контекста устройства используются для управления графическим выводом. Все подпрограммы рисования с GDI в качестве первого параметра принимают логический номер контекста устройства. Если Вы хотите воспользоваться графической библиотекой GDI для прорисовки в окне или вывода графики на принтер, Вы сначала должны получить логический номер контекста желаемого устройства.

Логические номера могут идентифицировать и другие объекты. Логическими номерами обладают объекты пользовательского интерфейса: меню, пиктограммы и курсоры. Логические номера могут идентифицировать и объекты, используемые при прорисовке - перья, кисти, шрифты, области и битовые образы. Даже динамически выделяемый участок памяти идентифицируется логическим номером.

Рассмотрим вкратце включаемый файл OWL.M, на который ссылается любая работающая с OWL программа.

Включаемые файлы OWL

Единственный включаемый файл OWL.H сводит воедино все включаемые файлы, необходимые для базового функционирования OWL. Рассматривая этот файл, Вы увидите, что он начинается двумя, на первый взгляд, непонятными строками:

```
#ifndef __OWL_H
#define __OWL_H
```

В конце же этого файла находится оператор:

```
#endif
```

Эти операторы гарантируют, что включаемый файл будет использован только однократно. В противном случае компилятор начнет сбиваться и "жаловаться". Такого рода операторы находятся во всех включаемых файлах OWL. Возможно, Вы захотите включить определения этого типа во все включаемые файлы, так как они снимут с Вас необходимость отслеживать избыточные операторы включения.

Ниже приводится список включаемых файлов, на которые ссылается OWL.H (прямо или косвенно).

Включаемый файл	Описание
-----------------	----------

<i>Applicat.H</i>	Определяет TApplication, класс "Прикладная программа" из OWL. Также содержит прототип для WinMain, являющийся точкой входа в любую прикладную программу
-------------------	---

<i>Dialog.H</i>	Определяет класс TDialog, класс блока диалога из OWL
-----------------	--

<i>MDI.H</i>	Определяет классы поддержки Многодокументного Интерфейса (Multiple Document Interface - MDI), TMDIClient, TMDIWindow
--------------	--

<i>Module.H</i>	Определяет TModule, базовый класс модуля из OWL, являющийся родительским классом для TApplication
<i>Object.H</i>	Определяет абстрактный базовый класс Object для библиотек OWL. Object является корневым в иерархии классов
<i>OwlDefs.H</i>	Определяет различные символические константы для использования библиотеками классов OWL
<i>WindObj.H</i>	Определяет класс TWindowsObject, базовый класс окна OWL, являющийся родительским для TWindow и TDialog
<i>Window.H</i>	Определяет класс TWindow, который может служить подклассом для создания класса главного окна прикладной программы
<i>Windows.H</i>	Включаемый файл библиотеки Windows. Этот файл (описываемый ниже) содержит определения типов, символические константы и определения структур данных, необходимые для соединения программы с Windows API (множественный Интерфейс Прикладных Программ Windows)

В следующих главах мы ближе рассмотрим классы, определяемые во включаемых файлах OWL. Сейчас же мы изучим содержимое включаемого файла WINDOWS.H, описывающего API в системе Windows.

Включаемый файл Windows

Если библиотеки языков C и C++ поставляются со множеством небольших включаемых файлов (STDIO.H, STRING.H и т.д.), то библиотеки Windows размещаются в одном большом (120 Кбайт) включаемом файле WINDOWS.H. На этот файл должен ссылаться (прямо либо косвенно) любой исходный файл, обращающийся к API системы Windows, поскольку в нем находятся соответствующие определения. В WINDOWS.H находятся три базовых типа определений: определения символических констант, типов данных и прототипов библиотечных функций. Давайте достаточно подробно рассмотрим каждое из этих определений.

Символические константы

В целом непосредственно помещать в программу логические номера, например 15 или 400, нецелесообразно, тогда Вашу программу становится трудно читать. Вместо них желательно использовать символические константы. По соглашению символические константы записываются целиком заглавными буквами, чтобы отличить их от имен переменных. В языке C символические константы создаются директивой препроцессора #define:

```
#define MAXOPENFILES 15
```

Поскольку WINDOWS.H используется для программ, написанных как на C, так и на C++, в нем используется только конструкция с #define вместо более усовершенствованного оператора const из C++.

В любом из этих языков символические константы улучшают читаемость программы и делают более удобным ее сопровождение. Вместо того чтобы выискивать по всему тексту программы непосредственно заданную константу, здесь достаточно в одной строке изменить значение символической константы.

В файле WINDOWS.H определено около 1500 символических констант. Чтобы легче было их рассортировать, применяется Венгерская Запись. Например, символические константы, описывающие оконные сообщения, начинаются с префикса WM_ (как в WM_CREATE и WM_DESTROY). Если константа используется единственной функцией, то префикс может быть сформирован из имени этой функции. Например, константа CW_USEDEFAULT используется исключительно функцией CreateWindow.

Определения типов данных

В файле `WINDOWS.H` определено всего несколько типов данных. Некоторые из них представляют собой не более чем просто удобный способ ссылки на распространенные в `C` типы. Например, в `WINDOWS.H` имеется следующий оператор:

```
typedef char far * LPSTR;
```

Это определение делает более простым определение дальних указателей на строку символов, поскольку

```
LPSTR lpszName;
```

эквивалентно

```
char far * lpszName;
```

Ниже приводится список широко распространенных типов данных.

Имя в <code>WINDOWS.H</code>	Определение в <code>C</code>
<code>BOOL</code>	<code>int</code>
<code>BYTE</code>	<code>unsigned char</code>
<code>DWORD</code>	<code>unsigned long</code>
<code>HANDLE</code>	<code>unsigned int</code>
<code>HDC</code>	<code>unsigned int</code>
<code>HWND</code>	<code>unsigned int</code>
<code>LONG</code>	<code>long</code>
<code>LPSTR</code>	<code>char far *</code>
<code>NPSTR</code>	<code>char near *</code>
<code>WORD</code>	<code>unsigned int</code>

Три типа описаны как `unsigned int`, это `HANDLE`, `HDC` и `HWND`. Каждый из них определяет логический номер. Выше в этой главе мы представили логический номер как способ идентификации объекта в `Windows`. Использование логических номеров позволяет скрыть от Вашей программы сложность объекта. Другими словами, `Windows` реализует инкапсуляцию данных. Объекты и логические номера объектов очень важны для программистов в `Windows`. Когда Ваша программа создает, например, окно, ему назначается некоторый логический номер, идентифицирующий это окно, это тип данных `HWND`.

Хотя работать с "сырыми" типами данных довольно тяжело, Вам, скорее всего, захочется максимально освоить записываемые заглавными буквами типы из `WINDOWS.H`. Дело в том, что они обеспечат переносимость Вашей программы в случае распространения API системы `Windows` на процессоры другой архитектуры. Если это случится, Вы можете рассчитывать на то, что фирма `Borland`, вероятнее всего, перенесет туда и свои библиотеки `OWL`, тогда программу можно будет в кратчайший срок запустить везде, где "живут" `Windows` и `OWL`.

В дополнение к простым типам данных `WINDOWS.H` содержит несколько определений структур. Понятно, что структура, описывающая прямоугольник, будет весьма полезна в среде, создающей прямоугольные окна. Ниже показана такая структура из `WINDOWS.H`:

```
typedef struct tagRect
{
    int left;
    int top;
    int right;
    int bottom;
} RECT;
```


Поскольку Windows позволяет выбирать объекты с помощью мыши, то, естественно, имеется и структура для записи позиции указателя мыши. Эта структура данных называется POINT:

```
typedef struct tdPOINT
{
    int Y;
    int X;
} POINT;
```

Прототипы функций

Важным свойством C++ является возможность создания прототипов функций. Прототипы обеспечивают средства, с помощью которых компилятор может автоматически выполнять некоторые критические операции по контролю ошибок. Прототип функции сообщает компилятору, как должна вызываться данная функция. Рассмотрим следующий прототип из WINDOWS.H:

```
BOOL FAR PASCAL TextOut(HDC, int, int, LPSTR, int);
```

Это объявление говорит компилятору, что подпрограмма обязательно должна вызываться с пятью параметрами. Если параметров меньше (или больше), то компилятор сообщает об ошибке. Если подумать, то одно это свойство оправдывает использование прототипов функций. Сколько раз в Вашей практике случалось, что Вы написали функцию, потом добавили в ее определение еще один параметр и забыли исправить те строки в программе, где происходит вызов этой функции? Не забудьте, что этим Вы вносили в свою программу ошибку. (И, конечно же, эта ошибка никак себя не проявляла, пока не приходило время показать программу начальству...) Прототип позволяет компилятору отлавливать и сообщать о такого рода ошибках, так что их легко найти и исправить.

Прототип сообщает компилятору об ожидаемых типах аргументов. В случае несовпадения переданных типов с типами в прототипе генерируется ошибка компилятора. Рассмотрим следующий вызов TextOut. Ясно, что четвертый параметр задан неверно:

```
TextOut (hDC, 10, 20, 30, 2);
```

Исходя из прототипа в WINDOWS.H, четвертый параметр должен быть дальним указателем на строку символов (char far *). Встретив в этом месте значение 30, компилятор сообщает о неверном типе параметра. Исправить ошибку в данном конкретном случае можно, взяв число в кавычки:

```
TextOut (hDC, 10, 20, "30", 2);
```

С помощью прототипов компилятор C или C++ может также контролировать правильность использования значения возврата функции. Речь идет опять о проверке несоответствия типов. Прототип для TextOut, например, определяет в WINDOWS.H значение возврата как BOOL. Следующие строки дадут ошибку компилятора:

```
char far * lpch;
lpch = TextOut (hDC, 10, 10, "Hello", 5);
```

Каждая функция Windows имеет прототип в WINDOWS.H, так что компилятор C может контролировать правильность их вызова. Использование прототипов полезно и при обращениях к библиотечным функциям Windows. К счастью, наличие прототипов является обязательным требованием к программам на C++. Иногда разработчики объединяют части программы, написанные на C и на C++. Это особенно важно при наличии больших наработок на C. В таких случаях рекомендуется создать прототипы и для функций на C, несмотря на то, что это не обязательное требование компилятора. Это сэкономит Вам массу времени.

Доступность прототипов функции привела к тому, что одна из ранее распространенных операций, а именно приведение типов, перешла в разряд устаревших. Однако большое число наработок было сделано пользователями до появления средства создания прототипов. Поэтому Вам следует просмотреть все старые программы для Windows, которые могут злоупотреблять явным приведением типов.

Устаревшая практика: приведение типов

Один из лучших способов изучения любой новой среды программирования состоит в том, чтобы взять и прочитать чужую программу. Однако нужно быть при этом осторожным, чтобы Вас не сбивала с толку встречающаяся в старых программах для Windows устаревшая практика приведения типов указателей. Это средство является устаревшим, поскольку новые компиляторы поддерживают прототипы функций, позволяющие ему автоматически генерировать правильный код.

Чуть выше мы рассматривали прототип функции TextOut:

```
BOOL FAR PASCAL TextOut(HDC, int, int, LPSTR, int);
```

Раньше Вы должны были бы записать примерно следующее обращение к этой функции:

```
TextOut (hDC, 10, 10, (LPSTR)"Всем привет", 12);
```

Отметим явное приведение к типу LPSTR. Тем самым Вы форсировали создание дальнего указателя. Дело в том, что в случае модели небольшой (small) или средней (medium) памяти выражение "Всем привет" вызвало бы генерацию ближнего (near) указателя. Но функция требует передачи ей в этом параметре дальнего указателя, поэтому необходимо было выполнить явное приведение типа указателя.

На первый взгляд, такое приведение типов не может причинить никакого вреда. Единственным недостатком кажется необходимость набирать лишние символы. Однако здесь имеется и реальная опасность. Следуя старой практике приведения типа каждого указателя, Вы спрячете некоторые проблемы, которые иначе бы компилятор мог обнаружить. Рассмотрим строку программы:

```
TextOut (hDC, 10, 10, (LPSTR)30, 2);
```

Возможно, программист в качестве четвертого параметра хотел задать строку 30. Если бы не было приведения типа, компилятор уведомил бы Вас об их несовпадении. Однако приведение принудительно устанавливает правильный тип данных, пряча ошибку от компилятора. Фактически это приведение говорит компилятору: "Я знаю, что я делаю. Прошу не задавать никаких вопросов".

Вывод ясен: избегайте приведения типов. Приведение отменяет автоматический контроль типов компилятором и скрывает от вас причину возможных ошибок.

Это вовсе не значит, что приведения вообще никогда не бывают нужны. Например, они часто требуются при работе с подпрограммами Windows, предназначенными для динамического выделения памяти. В их прототипах тип возврата объявлен как указатель на символьный тип. В случаях присвоения значения функции другому типу переменной компилятор выдаст соответствующее сообщение. Например, подпрограмма GlobalLock определена с типом возврата LPSTR(char far *):

```
LPSTR FAR PASCAL GlobalLock(HANDLE);
```

Следующие строки кода вызовут сообщение компилятора об ошибке:

```
int far * lpint; /* Определение указателя на целочисленный тип */  
lpint = GlobalLock(hMem); /* Здесь компилятор Вам "пожалуется" */
```

Вторая строка кода понадобилась бы при записи массива целочисленных значений в блок памяти, которому выделен логический номер hMem. Несмотря на сообщение компилятора об ошибке, эта строка дала бы правильный результат. Однако лучше, чтобы компилятор выдавал сообщения только в тех местах, где они действительно

но важны. Приведение типа в следующей строке даст тот же результат, но теперь ошибка несоответствия типов отсутствует:

```
lpint = (LPINT)GlobalLock(Mem);
```

(Разумеется, нужно использовать LPINT, а не int far *, так как это обеспечит переносимость Вашего кода.)

Итак, Вы видите, что иногда приведение типов просто необходимо. Однако компилятор, как правило, сам указывает, когда применение этой операции оправдано. Правило остается прежним: избегайте приведений типов. И только если компилятор начинает сообщать об ошибках, рассмотрите ситуацию и определите, поможет ли приведение типов решить эту конкретную проблему.

Сообщения

Прежде чем Вы начали писать объектно-ориентированные программы на C++, Вы, скорее всего, привыкли иметь дело со стилем программирования, в котором последовательность действий жестко определена в процедурах. Система Windows широко использует практику объектно-ориентированного программирования. Будьте осторожны, чтобы термин "сообщение" не ввел Вас в заблуждение. В контексте программы на C++ он фактически означает "вызов компонентной функции объекта". В контексте же Windows его смысл несколько иной.

Сообщения Windows генерируются в ответ на какие-либо изменения в пользовательском интерфейсе, например, перемещение окна или нажатие клавиши пользователем. Сообщение может уведомить программу об истечении некоторой выдержки времени. Они также используются в операциях совместного доступа к данным, подразумевающих действия с буфером вырезанного изображения Windows и Динамический Обмен Данными (Dynamic Data Exchange - DDE).

С точки зрения программирования сообщение представляет собой 16-битовое целое без знака, которому для лучшей читаемости присвоено параметрическое имя, начинающееся с WM_, например, WM_LBUTTONDOWN. Сообщение WM_LBUTTONDOWN означает, что пользователь нажал левую кнопку мыши. При отпускании этой кнопки направляется другое сообщение - WM_LBUTTONUP.

Очевидно, что сообщения Windows очень важны для программиста. Большая часть работы программиста в Windows и состоит в том, чтобы решить, какие сообщения нужно обрабатывать, а какие игнорировать. Как и сообщения в C++, сообщения в Windows не имеют заранее определенной последовательности.

В некоторой степени это просто вопрос конкретной реализации. Концептуально сообщения C++ и Windows одинаковы. Единственное их отличие состоит в том, что каждая функция C++ обрабатывает одно и только одно, сообщение. Напротив, единственная функция программы для Windows, называемая оконной процедурой, обрабатывает все сообщения, посылаемые данному типу окна. Начав работать с библиотекой классов C++, Вы узнаете об общедоступных (public) функциях (сообщениях) в различных классах. Аналогичным образом, учась работать с окнами Windows, Вы должны побольше узнать об общедоступных сообщениях. Чтобы помочь Вам приступить к их изучению, мы поместили в конце главы 5 и в приложении А раздел "Систематика Сообщений".

Теперь, рассмотрев некоторые базовые соглашения, принятые для программирования в Windows и C++, мы можем более подробно изучить исходный текст программы MIN. Начнем с частей MIN, касающихся класса объекта "Прикладная Программа", который называется TMinApplication, включая два класса, определяемых в OWL, - TModule и TApplication.

Глава 4. Объект "прикладная программа"

Возможно, Вы заметили, что программа MIN имеет два класса объектов: TMinApplication и TMinWindow. В этой главе мы обратим внимание на TMinApplication, объект "прикладная программа" для программы MIN. В следующей главе мы подробно рассмотрим класс объекта-окна TMinWindow.

Чтобы добраться до "сердца" TMinApplication, мы должны разобрать роль двух классов, на которых базируется класс "прикладная программа" MIN: это TApplication и его базовый класс TModule. В целом эти два класса отвечают за операции начальной загрузки и выхода при завершении прикладной программы. Они, кроме того, обеспечивают компонентные функции, перенаправляющие поток сообщений Windows стандартным образом. Ниже на рисунке показано место TMinApplication в иерархии классов.

Каждая прикладная программа для Windows, написанная на C++, C, Паскале или Ассемблере, имеет точку входа, которая называется WinMain. Поскольку это то самое место, где обычно программа для OWL создает свой объект "прикладная программа", мы начнем с того, что посмотрим, какие сведения WinMain может сообщить нам о прикладных программах для Windows.

Объявление процедуры WinMain

Процедура WinMain принимает четыре параметра, определенных следующим образом:

```
int PASCAL WinMain (HANDLE hInstance,  
                    HANDLE hPrevInstance,  
                    LPSTR lpszCmdLine,  
                    int nCmdShow)
```

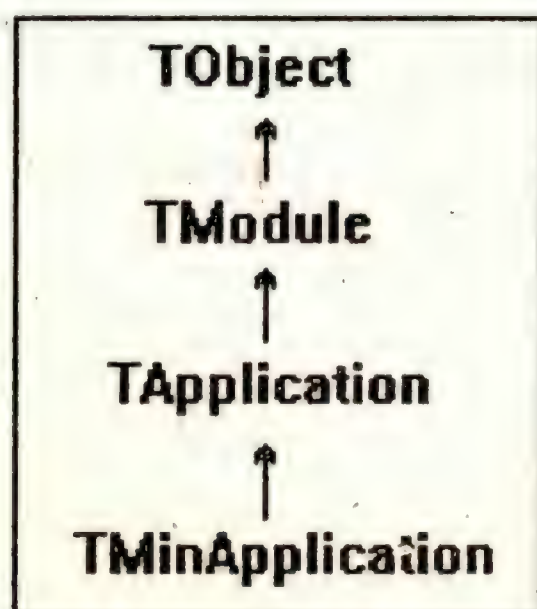


Рис.4.1. Позиция объекта "прикладная программа" программы MIN в иерархии объектов.

В начале работы программы система Windows передает информацию о ней и ее окружении. Эта информация содержится в параметре hInstance. При желании этот параметр можно считать именем программы. Windows присваивает каждой программе в системе уникальное имя. Разумеется, это имя не Джо или Фред, а 16-битовое целое без знака (тип, соответствующий типу HANDLE, определяемому в WINDOWS.H).

Система Windows сообщает программе ее имя, так как некоторые библиотечные подпрограммы Windows требуют передавать его в виде параметра. Это позволяет системе Windows знать, "кто вызывает". Хотя объект "прикладная программа" программы MIN не использует этот параметр, компонентные функции TWindow исполь-

зуют `hInstance` при создании главного окна нашей программы. Этот параметр используется, кроме того, компонентными функциями подклассов `MIN`, работающими с пиктограммой и курсором:

```
wc.hIcon=LoadIcon (wc.hInstance, "snapshot");  
wc.hCursor=LoadCursor (wc.hInstance, "hand");
```

Второй параметр - `hPrevInstance`, сообщает программе, с чем она связана. Если она не связана ни с одной текущей выполняемой программой, то значение `hPrevInstance` равно `NULL`. Каким образом Windows решает, что две программы связаны? По имени. Подобно тому, как люди с одинаковой фамилией (обычно) связаны друг с другом, так и две программы связаны, если одинаковы имена их файлов или имена их модулей. Вспомним, что файл определения модуля `MIN.DEF` содержит следующую строку:

`NAME MIN`

Когда пользователь выбирает программу для выполнения, система Windows сначала просматривает имена файлов на соответствие с именем текущей выполняемой программы. Затем Windows проверяет соответствие скрытого в исполняемом (`.EXE`) файле имени модуля с именем любой текущей выполняемой программы. Система Windows дает знать программе, что у нее имеется уже работающий в текущий момент "родственник", передавая ей его имя в параметре `hPrevInstance`.

Например, одновременно можно запустить четыре экземпляра программы `MIN`, три экземпляра `CLOCK`, два экземпляра `CALCULATOR` и один экземпляр `PAINTBRUSH`. При запуске каждого следующего экземпляра каждой программы `hPrevInstance` содержит имя предыдущего "члена семьи".

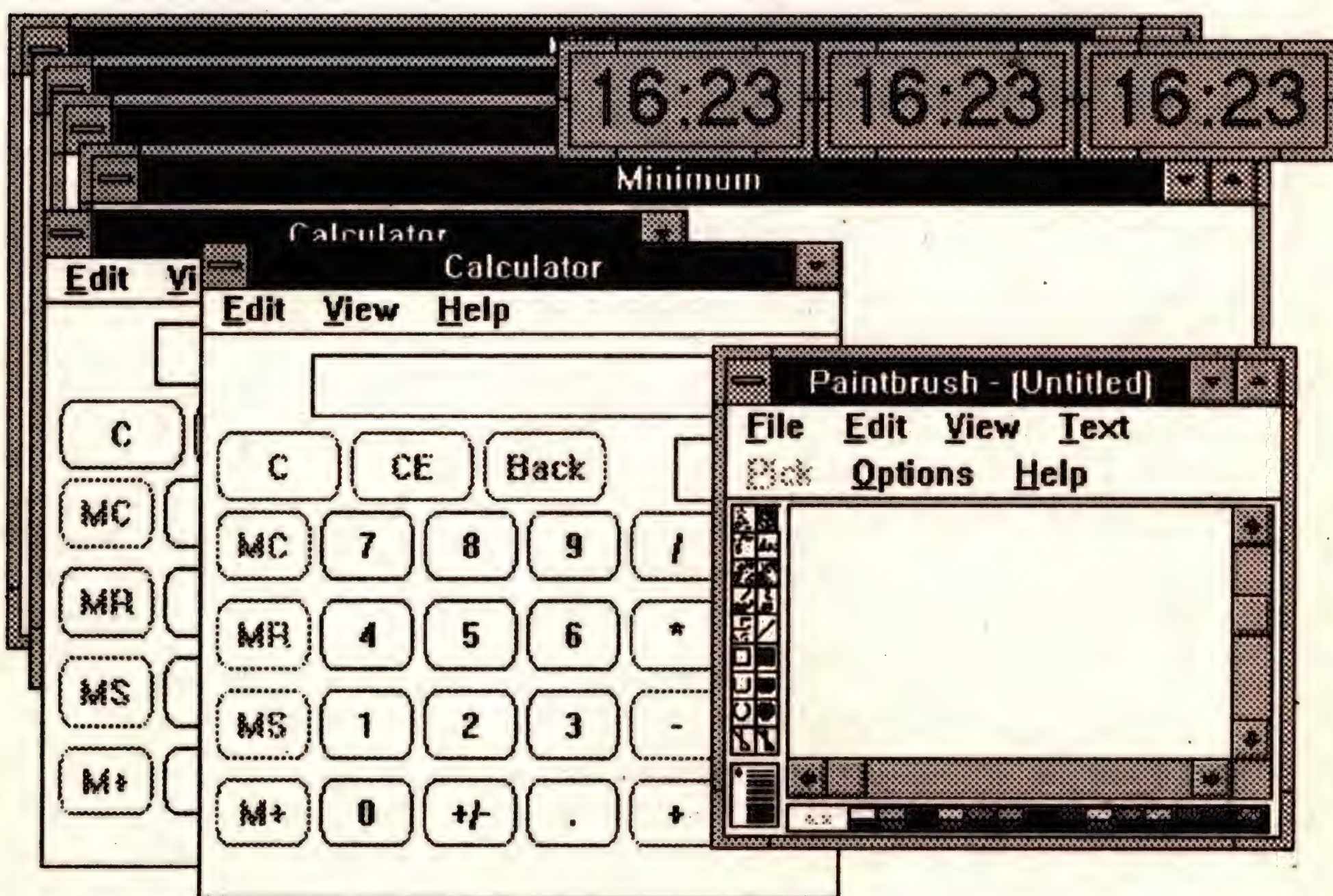


Рис.4.2. Четыре экземпляра программы MIN, три - CLOCK, два - CALCULATOR и один - PAINTBRUSH.

Зачем программе знать своих "родственников"? Дело в том, что, подобно связанным узами родства людям, родственники-программы объединены совместным владе-

нием. Например, для экономии памяти все экземпляры программы совместно используют один и тот же код и ресурсы. Естественно, единственное, чем они не могут пользоваться совместно, это данные, относящиеся к конкретным экземплярам. Программе нужно знать о наличии родственников для того, чтобы она могла вызывать их (то есть посылать им сообщения) с тем, чтобы расширить число совместно используемых вещей. Разумеется, это не является обязательным. Но если Вам хочется чем-то поделиться со своим кузеном Джимом, Вы получаете его номер телефона.

Третий передаваемый в WinMain параметр - `lpCmdLine`, дает аргументы командной строки для нашей программы. Эти аргументы могут создаваться множеством способов. Самый простой способ состоит в задании списка аргументов после имени программы в блоке диалога Properties ("Свойства") Менеджера Программ. Подобно параметру `argv` в точке входа в обычную программу на C или C++,

```
main (int argc, char **argv)
```

параметр `lpCmdLine` позволяет программе считывать из командной строки аргументы, такие как, например, имена файлов, значения ключей и т.д. Однако в отличие от `argv`, `lpCmdLine` указывает на единственную строку символов, а не на массив символьных указателей.

Последний параметр - `nCmdShow`, говорит программе, что делать с главным окном при ее первоначальном запуске. Должно ли оно быть минимизированным (до размера пиктограммы)? Или же оно должно выводиться в размер всего экрана? Рассматривая в следующей главе классы `TWindow` и `TWindowObject`, Вы увидите, что этот параметр передается объектом главного окна прикладной программы в библиотечную процедуру `Windows - ShowWindow`.

Теперь, понимая значение четырех параметров WinMain, мы можем перейти к изучению двух объектов, на которых построена наша прикладная программа: `TModule` и `TApplication`. Начнем с `TModule`.

Класс TModule

Поскольку Windows является многозадачной системой, одновременно в системе может находиться несколько выполняемых программ, например, MIN, CLOCK, CALCULATOR и PAINTBRUSH. С точки зрения OWL, каждая программа является модулем. В системе присутствует и еще ряд модулей, включая три главные библиотеки динамической компоновки Windows - `KERNEL.EXE`, `USER.EXE` и `GDI.EXE`, а также библиотеку динамической компоновки OWL `OWL.DLL`. Кроме того, другие модули включают в себя библиотеки динамической компоновки драйверов устройств: `DISPLAY.DRV`, `MOUSE.DRV`, `KEYBOARD.DRV` и т.д.

Система Windows знает о двух типах модулей: прикладных программах и библиотеках динамической компоновки (DLL). Класс `TModule` отражает способ представления модуля Windows в библиотеке классов OWL. При создании любого типа модуля с использованием OWL Вы начинаете с того, что создаете объект `TModule`. Рассмотрим поближе класс `TModule`. Как определено в `MODULE.H`, в `TModule` входят следующие компонентные данные:

Тип	Имя	Описание
<i>HANDLE</i>	<code>hInstance</code>	Копия параметра WinMain с логическим номером экземпляра прикладной программы
<i>LPSTR</i>	<code>lpCmdLine</code>	Копия параметра WinMain для командной строки прикладной программы
<i>int</i>	<code>Status</code>	Флаг ошибки
<i>LPSTR</i>	<code>Name</code>	Имя прикладной программы

Для доступа к любому из перечисленных элементов данных Вы должны взять указатель на Ваш объект "прикладная программа" (который, вообще говоря, являет-

ся потомком класса Module). Это можно сделать, вызвав GetApplicationObject(). Например, для получения логического номера экземпляра прикладной программы

```
HANDLE hInstance;  
PTApplication ptApplication;  
  
ptApplication = GetApplicationObject();  
hInstance = ptApplication->hInstance;
```

Вам может понадобиться копия hInstance для доступа к ресурсам или для создания объектов пользовательского интерфейса, хранимых в виде ресурсов - меню, блоков диалога, курсоров, пиктограмм и т.п.

Компонент данных lpCmdLine представляет собой копию третьего передаваемого в WinMain параметра.

Компонент Status используется объектом "прикладная программа" для хранения условий, вызвавших ошибку во время инициализации, либо для хранения кода возврата прикладной программы при ее завершении.

Компонент данных Name содержит указатель на ASCII-строку, передаваемую нашей программой конструктору объекта "прикладная программа". Поскольку это поле доступно любой части нашей прикладной программы, то в нем удобно держать строку, которая могла бы быть использована в блоках сообщений и диалогов. Такое повсеместное использование одной и той же копии строки с именем программы уменьшает расход памяти, выделяемой для хранения строк символов OWL-программы.

Класс TModule имеет 15 компонентных функций. Полное их описание не является целью этой книги, так как главным образом они предназначены для поддержки внутренних операций объекта-модуля.

Имя функции	Описание
TModule()	Конструктор TModule
TModule()	Деструктор TModule
Error()	Функция для сообщения об ошибке
ExecDialog()	Создает модальный блок диалога с обработкой ошибок
GetClientHandle()	Часть поддержки многодокументного интерфейса (MDI) OWL. Возвращает логический номер клиентского (проблемного) окна MDI
GetParentObject()	Возвращает указатель на оконный объект OWL по логическому номеру окна Windows. Если указатель недоступен, то он будет создан
hashValue()	Виртуальная функция, определяющая кеш-значение для данного класса
isA()	Виртуальная функция, определяющая класс данного объекта.
isEqual()	Виртуальная функция, позволяющая проверить, адресуют ли две ссылки один и тот же объект
LowMemory()	Проверяет, была ли начата работа с резервным пулом памяти
MakeWindow()	Создает окно или немодальный блок диалога
nameOf()	Виртуальная функция, позволяющая определить имя класса в виде текстовой ASCII-строки
printOn()	Виртуальная функция, позволяющая вывести на экран информацию об объекте для отладки
RestoreMemory()	Увеличивает размер области динамической памяти
ValidWindow()	Проверяет действительность указателя на оконный объект

Подпрограмма MakeWindow реализует первичный механизм OWL создания окон Windows. Эта компонентная функция будет подробнее рассмотрена в следующей

главе, когда мы будем изучать класс TWindow. Подпрограмма MakeWindow, кроме того, создает немодальные блоки диалога. Связанная с ней функция ExecDialog создает модальные блоки диалога. В главе 14 функции, используемые для работы с блоками диалога, рассматриваются более детально.

Большая часть работы на интересующем нас уровне прикладной программы выполняется производным от TModule классом, который называется TApplication. Рассмотрим его.

Класс TApplication

Класс TApplication является непосредственным предком объекта "прикладная программа" MIN. Класс TApplication занимается вещами, которые обычно делает прикладная программа для Windows: инициализирует и создает главное окно, а затем опрашивает системные сообщения. Посмотрим, как работает TApplication и что он в совокупности с библиотечными функциями Windows способен Вам дать.

Ниже приведены компонентные данные TApplication, определяемые в APPLICAT.H:

Тип	Имя	Описание
HANDLE	hPrevInstance	Копия параметра WinMain с логическим номером предыдущего экземпляра
int	nCmdShow	Копия параметра WinMain, задающая способ открытия главного окна программы. Передается подпрограмме Windows ShowWindow
PTWindowsObject	MainWindow	Указатель на оконный объект OWL, являющийся главным окном прикладной программы
HANDLE	HAccTable	Логический номер таблицы кодовых комбинаций клавиш, если программа имеет такую таблицу
PTWindowsObject	KBHandlerWnd	Указывает на TWindowsObject, который должен принимать ввод с клавиатуры

Первые два элемента списка hPrevInstance и nCmdShow являются копиями второго и четвертого параметров, передаваемых в WinMain. Прикладная программа обычно использует их только в момент начальной загрузки.

Компонент данных MainWindow представляет собой указатель на оконный объект. С точки зрения программирования в Windows это имеет смысл, так как практически каждая программа для Windows имеет только одно "главное" окно. При закрытии главного окна пользователь ожидает завершения и соответствующей прикладной программы. А компонент TApplication обеспечивает аналогичное поведение программ OWL.

Компонент данных HAccTable предназначен для логического номера таблицы кодовых комбинаций клавиш. Такая таблица содержит команды клавиатуры. Например, элемент этой таблицы может определять преобразование комбинации нажатий Shift+Ins в команду Edit. Paste для использования при работе с карманом Windows. Более подробно таблицы кодовых комбинаций клавиш рассматриваются в главе 11, где Вы увидите, как родительский класс TApplication упрощает поддержку кодовых комбинаций клавиш.

Компонент данных KBHandlerWnd указывает на активный оконный объект OWL. Активным называется окно, с которым в текущий момент работает пользователь. Это может быть немодальный блок диалога, главное окно прикладной программы или окно кадра MDI. Изменяя цвет заголовка или границ, окно показывает пользователю, что оно является активным. Библиотека OWL использует этот компонент данных для поддержки ввода с клавиатуры. Например, с помощью клавиши Tab пользователь может проходить между порожденными окнами внутри текущего активного окна.

Ниже приводятся компонентные функции TApplication:

Имя функции	Описание
PUBLIC (общие): TApplication()	Конструктор TApplication
TApplication()	Деструктор TApplication
CanClose()	Опрашивает допустимость закрытия прикладной программы. Унифицирует запрос, необходимый типичной программе под Windows для контроля двух сообщений Windows: WM_CLOSE и WM_QUERYENDSESSION. Объект "прикладная программа" опрашивает главное окно прикладной программы, которое, в свою очередь, опрашивает каждое из порожденных окон, чтобы гарантировать, помимо всего прочего, что данные пользователя не будут потеряны
isA()	Виртуальная функция, позволяющая определить класс данного объекта
nameOf()	Виртуальная функция, позволяющая определить имя класса в виде текстовой ASCII-строки
Run()	Главная функция прикладной программы, заставляющая объект "прикладная программа" выполнить все операции инициализации и обработки, которые нужны для выполнения программы в Windows
SetKBHandler()	Определяет оконный объект, который будет обрабатывать ввод с клавиатуры
PROTECTED (защищенные): IdleAction()	Подфункция цикла приема сообщений, реализующая фоновую обработку
InitApplication()	Подпрограмма инициализации первого экземпляра прикладной программы
InitInstance()	Подпрограмма инициализации каждого экземпляра прикладной программы
InitMainWindow()	Подпрограмма инициализации главного окна прикладной программы
MessageLoop()	Функция цикла приема сообщений прикладной программы
ProcessAccels()	Подфункция цикла приема сообщений, реализующая обработку кодовых комбинаций клавиатуры
ProcessAppMsg()	Подфункция цикла приема сообщений, реализующая задачи, специфичные для данной прикладной программы
ProcessDlgMsg()	Подфункция цикла приема сообщений, реализующая обработку ввода с клавиатуры для немодальных блоков диалога
ProcessMDIAccels()	Подфункция цикла приема сообщений, реализующая обработку ввода с клавиатуры для окон с многодокументным интерфейсом (MDI)

Работа TApplication организуется одной компонентной функцией TApplication::Run, определяемой таким образом:

```
void TApplication::Run()
{
    if ( !hPrevInstance )
        InitApplication();
    if ( Status == 0 )
        InitInstance();
    if ( Status == 0 )
        MessageLoop();
    else
        Error(Status);
}
```


При всей своей лаконичности функция Run имеет все четыре составляющие, необходимые для выполнения программы в Windows:

1. Выполнение инициализации на уровне прикладной программы, т.е. инициализация первого экземпляра MIN
2. Выполнение инициализации на уровне экземпляра, т.е. выполнение всех действий, которые обязательны для каждого экземпляра MIN
3. Вхождение в краткий цикл опроса сообщений от пользователя. В этом цикле программа находится все время, после того как завершена инициализация
4. Завершение, если при инициализации произошла ошибка

Из этих четырех компонентных функций наибольший интерес представляют вторая и третья, поэтому мы рассмотрим их подробнее. Первая функция - InitApplication() просто резервирует место для определения кода инициализации первого экземпляра. Последняя функция - Error() выдает блок с сообщением с обозначенным кодом ошибки и завершает работу программы. На этапе разработки OWL-программы этого вполне достаточно, а для конечного продукта обработку ошибок можно сделать более сложной.

На следующем рисунке показана иерархия вызываемых с помощью TApplication::InitInstance функций. Эта схема поможет Вам понять, что реально происходит, поскольку вызовы проходят как вверх, так и вниз по иерархии объекта, и даже по иерархии другого класса объекта (TWindow/TWindowObject). В этой схеме вызов библиотечной функции показан в виде компонентной функции воображаемого класса Win3.

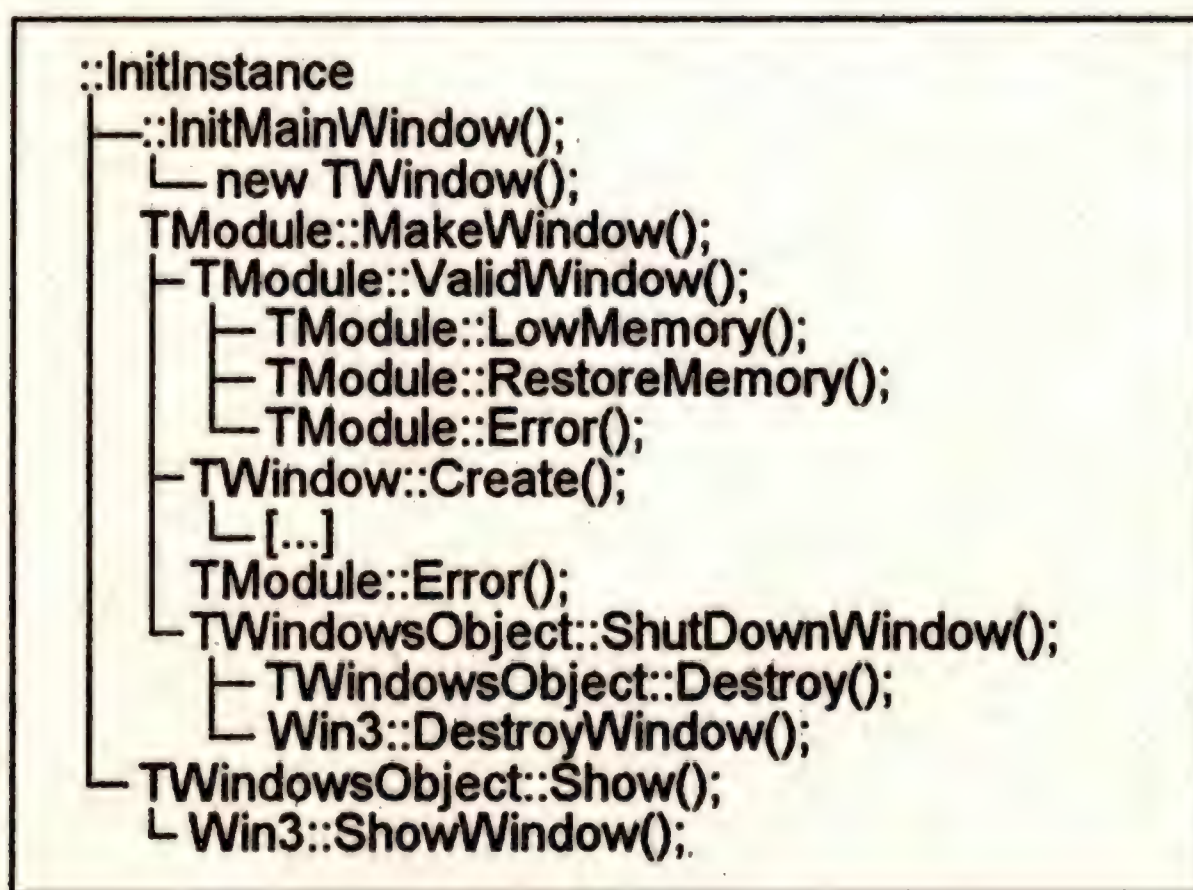


Рис.4.3. Инициализация объекта "прикладная программа", выполняемая компонентной функцией класса TApplication::InitInstance.

Функция InitInstance начинает работать с вызова InitMainWindow. Большинство прикладных программ OWL переопределяет эту компонентную функцию, так как по умолчанию она создает пустой объект TWindow. Чтобы "вдохнуть жизнь" в свою прикладную программу, Вы должны определить класс - потомок от TWindow. В программе MIN эту роль играет класс TMinWindow. Затем программа MIN создает экземпляр нового оконного объекта в своей функции InitMainWindow:

```

void TMinTApplication::InitMainWindow ()
{
    MainWindow = new TMinWindow (NULL, "Minimum", NULL);
}
  
```


После того как оконный объект OWL создан конструктором, компонентная функция `TModule::MakeWindow` создает окно `Windows`. Эта компонентная функция действует подобно коммутатору, связывая различные компонентные функции оконного объекта с библиотеками `Windows`. По ходу дела она выполняет ряд действий по контролю ошибок, проверяя наличие достаточного объема памяти и правильность создания окна. Фактическое создание окна производится компонентной функцией оконного объекта `Create`.

Когда созданы оконный объект и окно, в качестве последнего шага процесса инициализации нужно сделать окно видимым. Эту задачу выполняет компонентная функция `TWindowsObject::Show()`. Как показано на схеме, она вызывает библиотечную функцию `Windows ShowWindow`, передавая ей параметр `nCmdShow` из функции `WinMain` нашей программы. Это позволяет `MIN` открыть окно нужного размера: минимизированное (до размеров пиктограммы), максимизированное (до размеров экрана) или нормального размера. На следующем рисунке показана иерархия функций, вызываемых подпрограммой обработки сообщений нашего объекта "прикладная программа" `TApplication::MessageLoop`. Библиотечные подпрограммы `Windows` снова показаны как принадлежащие воображаемому объекту класса `Win3`.

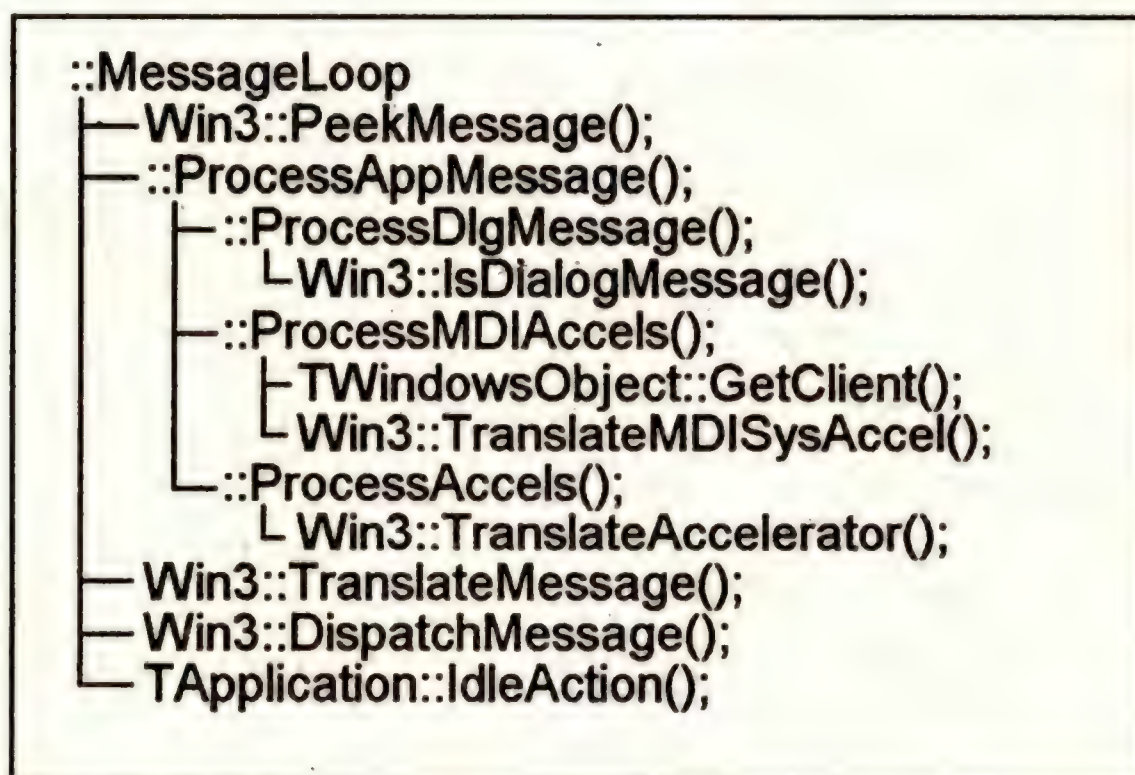


Рис.4.4. Цикл опроса сообщений объекта "прикладная программа" в `TApplication::MessageLoop`.

Войдя один раз в цикл опроса сообщений, программа для `Windows` непрерывно считывает сообщения и передает их в соответствующий обработчик. В цикле опроса сообщения обычно не учитывается их содержание, предоставляя обработку оконным объектам. Исключение здесь составляют три типа объектов пользовательского интерфейса: немодальные блоки диалога, окна многодокументного интерфейса и кодовые комбинации клавиатуры. Они представлены тремя подфункциями цикла опроса сообщений: `ProcessDlgMessage`, `ProcessMDISysAccels` и `ProcessAccels`. Подфункцию `ProcessDlgMessage` мы будем подробно рассматривать в главе 14, где говорится о немодальных блоках диалога. Обработка кодовых комбинаций клавиатуры изучается в главе 11.

Прежде чем рассматривать прохождение сообщений с точки зрения `MessageLoop`, попытаемся понять, как программа `MIN` использует класс `TApplication` для создания собственного объекта "прикладная программа".

Класс `TMinApplication` программы `MIN`

Жизнь любой программы для `Windows` начинается с функции `WinMain`. Большинство OWL-программ имеет функцию `WinMain`, которая выглядит, как в `MIN`:

```

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
  
```



```

{
    TMinApplication Min ("MIN", hInstance, hPrevInstance,
                        lpzCmdLine, nCmdShow);
    Min.Run();
    return Min.Status;
}

```

Объект "прикладная программа" создается и начинает работу.

Программа MIN использует класс TApplication, как базовый, для собственного класса TMinApplication. Этот класс имеет две компонентные функции: конструктор и функцию, переопределяющую аналогичную функцию родительского класса:

```

class TMinApplication : public TApplication
{
public:
    TMinApplication(LPSTR AName);
    virtual void InitMainWindow();
};

```

Функция-конструктор TMinApplication не выполняет никаких реальных действий. Она просто резервирует это имя для программ, которые будут использовать MIN в качестве своей отправной точки.

Почти каждая OWL-программа переопределяет InitMainWindow. В MIN это выглядит так:

```

void TMinApplication::InitMainWindow()
{
    MainWindow new TMinWindow (NULL, "Minimum");
}

```

Она создает оконный объект. Если не переопределять InitMainWindow, то оригинальная ее версия создаст объект TWindow. Такое окно будет пустым, хотя будет работать правильно: будет работать его системное меню, которое позволит изменять свои границы, а также максимизировать и минимизировать их.

Когда объект "прикладная программа" MIN выполнит собственное создание, он создаст оконный объект, выведет окно на экран, а затем войдет в цикл опроса сообщений. По завершении инициализации основная работа объекта "прикладная программа" состоит в том, чтобы обеспечивать передачу сообщений между пользователем и оконным объектом.

Рассмотрим поближе сообщения, которые фактически играют роль крови каждой программы для Windows. Может показаться, что мы затронем проблему, которая пока будто бы не к месту - свойства многозадачности Windows. Однако многозадачность и сообщения тесно взаимосвязаны, так как обе эти проблемы имеют дело с механизмом доставки сообщений Windows. Начнем изучать сообщения, принимая во внимание взаимосвязь сообщений со свойством многозадачности Windows.

Сообщения: механизм ввода в программу и квантование времени для реализации многозадачности.

В главе 1 мы ввели идею о том, что Windows представляет собой многозадачную операционную систему. Однако в отличие от других многозадачных систем Windows сама не прерывает одну программу, чтобы дать поработать другой. Подход к многозадачности на базе прерываний иногда называют многозадачностью с вытеснением. Напротив, Windows является многозадачной системой без вытеснения. Это означает, что программа прерывается не по инициативе операционной системы, а прерывает сама себя, чтобы дать возможность поработать другой программе. Как организована такого рода работа?

Переключатель задач Windows встроен в ее систему доставки сообщений. Программы для Windows принимают ввод в программу от пользователя и объектов пользовательского интерфейса посредством сообщений. Некоторые из этих сообщений

связаны с аппаратным обеспечением и сообщают программе о нажатии пользователем клавиши или кнопки мыши. Другие исходят от объектов пользовательского интерфейса, таких как окна, меню и полосы прокрутки. Для доступа к потоку сообщений программа обращается к системе доставки сообщений. За одно обращение программа получает одно сообщение. Если сообщений для программы больше нет, то система доставки начинает доставлять сообщения той программе, для которой таковые имеются.

Из сказанного ясно, что сообщения в Windows имеют более широкий смысл, чем сообщения в реальном мире, т.е. они служат не только для связи. Для программы в Windows сообщение является единицей обработки. Они имеют эффект создания квантов времени, в течение которого может выполняться данная программа. Чтобы лучше понять двойственную роль сообщений, рассмотрим поближе систему доставки сообщений.

Программа для Windows получает сообщения двумя способами. Первый из них состоит в чтении буферов сообщений. Система Windows не позволяет выполнять прямой доступ к своим внутренним буферам, но сама выполняет для нас чтение буфера при вызове программой одной из двух библиотечных подпрограмм: GetMessage или PeekMessage. (Между этими функциями имеются тонкие различия, но сейчас они для нас не важны). Вспомните, вторая из них вызывается компонентной функцией MessageLoop класса TApplication.

Происходит чтение двух системных буферов: буфера очереди аппаратных событий и буфера очереди программных событий. Очередь аппаратных событий содержит глобальные события системы, связанные с нажатиями клавиш и кнопки мыши. Очередь программных событий содержит сообщения, специфичные для прикладной программы. Каждая программа имеет свою очередь сообщений. Более того, каждый экземпляр программы имеет свою очередь сообщений. Если в очереди обнаружено сообщение для программы, то GetMessage передает программе это сообщение.

Если GetMessage не находит сообщений для программы, то она "укладывает программу спать". В это время переключатель задач Windows берет управление и передает его другой программе, для которой сообщения имеются. Итак, подпрограмма GetMessage обеспечивает функционирование многозадачной системы, управляемой событиями.

Второй способ получения программой сообщений состоит в непосредственном вызове одной из ее оконных процедур. В терминах C++ оконная процедура - это компонентная функция для любых сообщений, посылаемых данному классу окон. Оконная процедура вызывается так, как если бы она была подпрограммой самой Windows. Фактически Вы можете представлять себе оконную процедуру как устанавливаемое расширение Windows. Доставляемые таким образом сообщения не буферизуются в очереди, а обрабатываются непосредственно в момент вызова.

Остановимся на минуту и рассмотрим все аспекты, связанные с этими двумя подходами. Первый метод, в котором буферы сообщений для нас считываются функциями GetMessage или PeekMessage, называется активной моделью (pull-model) обработки сообщений. Второй метод называется пассивной моделью (push-model) обработки сообщений и подчеркивает пассивную роль программы, которая ожидает вызова, предоставляя Windows поместить сообщение в нашу оконную процедуру.

Может показаться странным, что Windows обладает двумя моделями доставки сообщений. В конце концов, Apple Macintosh обходится одним активным механизмом, а Xerox Star - одним пассивным механизмом доставки. Зачем же иметь два? Немного истории, и Вам все станет ясно.

Прежде чем начать распространение Windows в ноябре 1985 года, фирма Microsoft два года разрабатывала эту систему. Ранние внутренние версии Windows (1983-1984) строились целиком на пассивном механизме приема сообщений. Тогда Windows была просто пакетом подпрограмм. Для каждого оконного класса было определено по 10 компонентных функций, по одной для каждого из 10 различных типов событий пользовательского интерфейса. Например, одна подпрограмма вызыва-

лась при создании окна, другая доставляла ввод с клавиатуры или мыши, а третья служила для прорисовки в окне.

Все было прекрасно, за исключением ввода с клавиатуры и мыши. Инженеры Microsoft никак не могли совместить управляемую событиями аппаратную часть с механизмом пассивного приема этих событий для обработки. Кроме того, механизм пассивного приема работает в очень упорядоченном, синхронизированном режиме. С другой стороны, управляемая событиями аппаратура создает довольно-таки хаотичную картину. В зависимости от того, как работает пользователь: выполняет набор с клавиатуры со скоростью 60 ударов в минуту, работает с функциональными клавишами или же "мучает мышку" — ввод поступает в произвольном порядке через непредсказуемые интервалы времени.

В случае чисто пассивной модели каждой прикладной программе требуется критичный по времени обработчик прерывания. В противном случае, если обработчик будет работать с одним событием клавиатуры или мыши слишком долго, то следующее может быть потеряно, так как пришедшее слишком рано аппаратное прерывание прервет обработку другого до того, как она будет доведена до конца. В том и другом случае события клавиатуры и мыши будут обработаны неправильно. Где решение? Система Windows использует обе — активную и пассивную модели приема сообщений.

Преимущество пассивной модели обработки сообщений состоит в том, что она разрешает объектам пользовательского интерфейса, являющимся частью операционной системы, напрямую взаимодействовать с прикладными программами. Можно ли представить себе более тесную связь с операционной системой, чем ту, при которой Ваша программа как бы становится ее подпрограммой?

В случае активной модели приема, напротив, непредсказуемые аппаратные события хранятся в системном буфере. Однако активная модель позволяет создавать модальные циклы опроса, в которых программа сама может отфильтровывать сообщения, которые она не должна обрабатывать. Простой пример модального цикла в обычных не относящихся к Windows программах, когда выдается следующее сообщение:

Strike Any Key When Ready ...

("При готовности нажмите любую клавишу")

и выполнение программы приостанавливается, пока пользователь не начнет ввод с клавиатуры.

Совместное использование активной и пассивной моделей дало уникальную гибридную систему в Windows. Вернемся к исходному коду цикла опроса сообщений TApplication, чтобы разобраться в деталях данного гибридного метода.

Стандартный цикл опроса сообщений

Каждая программа в Windows имеет цикл опроса сообщений. Тем самым она получает сообщения, жизненно важные для ее правильного функционирования. Стандартный цикл опроса сообщений программы под Windows имеет следующий вид:

```
while (GetMessage(&msg, 0, 0, 0))
{
    TranslateMessage(&msg); /* Ввод с клавиатуры */
    DispatchMessage(&msg);
}
```

Данный цикл работает в течение всего времени выполнения программы. Каждая итерация представляет собой прием одного сообщения из буфера сообщений. За исключением WM_QUIT, каждое сообщение заставляет GetMessage вернуть значение TRUE. Принимая WM_QUIT, программа выходит из цикла опроса сообщений и завершает свою работу. Обсуждая оконный объект MIN в главе 5, Вы увидите, как код

в классе TWindowsObject при разрушении главного окна прикладной программы посылает сообщение WM_QUIT. Без этого сообщения программа не могла бы правильно выполнить выход в систему, а занимаемая ею память не была бы возвращена системе и потеряна для других программ.

Рассмотрим каждую подпрограмму этого небольшого цикла. Только что мы высказали мысль, что GetMessage принимает сообщения в нашу программу из системных буферов - очереди аппаратных событий и очереди программных событий. Мы можем заставить GetMessage фильтровать события, задав для этого фильтра критерии во втором, третьем и четвертом параметрах. Сейчас нас интересуют все сообщения, т.к. фильтрация отменена путем установки параметров в ноль.

Больше всего нас интересует первый параметр, указатель на структуру, которую GetMessage заполняет данными о сообщении. Отметим, что указатель на эту структуру также является единственным параметром при вызове остальных подпрограмм в цикле опроса сообщений.

Компонентная функция TApplication MessageLoop() определяет локальную переменную для хранения данных о сообщении следующим образом:

MSG Message;

Определение MSG находится в WINDOWS.H и имеет следующий вид:

```
typedef struct tagMSG
{
    HWND hwnd;
    WORD message;
    WORD wParam;
    LONG lParam;
    DWORD time;
    POINT pt;
} MSG;
```

Рассмотрим каждый из шести элементов структуры:

- * hwnd - целое без знака, содержит логический номер окна. Каждому окну в системе присваивается уникальный логический номер. Подобно описанному выше логическому номеру экземпляра, логический номер окна - это просто присвоенное окну имя. Причина того, что в структуре сообщения находится номер окна, состоит в том, что каждое сообщение всегда адресовано тому или иному окну.

- * message - целое без знака, хранит тип сообщения, закодированного 16-битовым значением. Разные типы сообщений Windows рассматриваются ниже (см. главу 5).

- * wParam и lParam содержат данные о сообщении. Смысл их зависит от типа сообщения. Например, сообщение команды меню использует wParam для идентификации выбранного элемента меню. Сообщения мыши помещают в lParam координаты указателя мыши. Элемент wParam имеет размер целого без знака, а lParam - длинного целого без знака. Это позволяет записывать шесть байтов данных для каждого сообщения.

- * Элементы time и pt редко используются программами для Windows. Оба описывают состояния системы в момент постановки сообщения в очередь: time хранит время, а pt - позицию указателя мыши в этот момент.

Второй вызываемой в обычном цикле опроса сообщений функцией является TranslateMessage. Эта функция вызывает драйвер клавиатуры Windows для преобразования "необработанных" сообщений-нажатий клавиш (WM_KEYDOWN) в ASCII-значения, которые ставятся в очередь программных событий в виде сообщений WM_CHAR. Это позволяет программе отличить, например, "А" от "а" без анализа состояния клавиши регистра.

Функция TranslateMessage также обеспечивает поддержку международных клавиатур. Конечно, для обозначения суммы в 35 фунтов стерлингов английский пользователь захочет записать это значение со знаком фунта. Если при этом он пользуется

текстовым редактором, работающим в Windows, то этот редактор вызовет TranslateMessage для получения правильного символа. Функция TranslateMessage позволяет применять диакритические знаки для написания некоторых букв французского, немецкого, испанского и других языков, где такие знаки используются.

Последняя процедура цикла - DispatchMessage берет данные о сообщении из структуры message и использует их для вызова соответствующей оконной процедуры. Она помещает сообщение в эту оконную процедуру для обработки.

Вы можете спросить, а можно ли ускорить дело, непосредственно вызывая оконную процедуру без DispatchMessage? Ответ однозначный - нельзя. Все вызовы оконных процедур должны исходить от Windows.

Суть проблемы связана со способом адресации данных. Программа для Windows обычно имеет свой собственный сегмент данных. Библиотека динамической компоновки, управляющая пользовательским интерфейсом Windows USER, тоже имеет собственный сегмент данных. Когда USER вызывает оконную процедуру, она помещает идентификатор сегмента данных оконной процедуры в регистр AX процессора.

В начале оконной процедуры компилятор генерирует следующий код:

```
PUSH DS      ; Сохранить значение сегмента данных USER
MOV DS,AX    ; Инсталлировать там значение нашего сегмента данных
```

В самом конце работы оконной процедуры, перед возвратом, мы восстанавливаем сегмент данных модуля USER:

```
POP DS      ; Восстановить сегмент данных USER
```

Для того чтобы программа для Windows напрямую могла вызвать оконную процедуру, она должна иметь возможность записать в регистр AX правильное значение сегмента данных. В противном случае оконной процедуре не будут доступны данные из ее собственного сегмента данных.

Цикл опроса сообщений OWL

Сначала мы продемонстрировали Вам стандартный цикл опроса сообщений, поскольку Вы встретите его в других программах для Windows. Однако в OWL принят другой цикл опроса сообщений. Причина состоит в том, что OWL предоставляет Вам как бы шаблон для построения более сложных задач. Ниже показана компонентная функция MessageLoop() из класса TApplication:

```
void TApplication::MessageLoop()
{
    MSG Message;
    while ( TRUE )
    {
        if ( PeekMessage(&Message, 0, 0, 0, PM_REMOVE) )
        {
            if ( Message.message == WM_QUIT )
                break;
            if ( ProcessAppMsg(&Message) )
            {
                TranslateMessage(&Message);
                DispatchMessage(&Message);
            }
        }
        else // Ожидающих обработки сообщений нет
            IdleAction();
    }
    Status = Message.wParam;
}
```


Хотя показанный цикл выглядит больше стандартного, по многим компонентам они совпадают. Сообщения принимаются при помощи PeekMessage, поддержка клавиатуры обеспечивается TranslateMessage, выбор оконной процедуры и передача в нее сообщения выполняются с помощью DispatchMessage. Как ранее упоминалось, ProcessAppMsg поддерживает немодальные блоки диалога, кодовые комбинации клавиш и окна многодокументного интерфейса.

Циклы отличаются вызовом IdleAction. Данная виртуальная компонентная функция базового класса TApplication просто резервирует это имя для последующего переопределения, позволяя "украсть" у системы циклы фоновой обработки. Как упоминалось выше, Windows - это многозадачная система без вытеснения. Следовательно, Вы не можете (сейчас) создать фоновую цепочку для организации фоновой обработки. Однако, когда прикладная программа принимает сообщения с помощью PeekMessage, появляется такая возможность. Если очередь сообщений пуста, то функция возвращает FALSE. Программы для Windows, которым требуется выполнять фоновую обработку, часто используют цикл опроса сообщений, аналогичный показанному в приведенном выше фрагменте. Отметим, что цикл опроса сообщений OWL требует специальным образом обрабатывать сообщение WM_QUIT, поскольку PeekMessage не реагирует на это сообщение так, как это делает GetMessage.

Для того чтобы задать фоновую обработку в цикле, выполняемую, пока очередь сообщений пуста, нужно определить подпрограмму IdleAction. Эта подпрограмма позволит Вам пересчитать электронную таблицу, опросить устройство, расставить страницы на документе или выполнить любые другие действия, не требующие прямого участия пользователя. Поскольку используется механизм диспетчеризации заданий без вытеснения, следует проверить, не задаете ли Вы слишком много времени на выполнение этой подпрограммы. Рекомендуется разбить Ваш фоновый процесс на несколько небольших хорошо управляемых фрагментов. Конечно, не всякая программа в среде Windows требует фоновой обработки. Вы можете просто игнорировать эту возможность и разрабатывать программы, для которых квантование времени производится по поступлении обычных сообщений. Мы завершаем обсуждение объекта "прикладная программа" программы MIN. В процессе этого обсуждения мы затронули некоторые основы архитектуры самой системы Windows. Однако на этом наше "путешествие" не закончилось: нам предстоит рассмотреть объект окна пользовательской программы. Этим мы займемся в следующей главе.

Глава 5. Классы оконных объектов из библиотеки OWL

Кроме объектов "прикладная программа", программам из библиотеки OWL требуются оконные объекты. Библиотека OWL поддерживает оконные объекты с помощью расширяемых классов, которые упрощают взаимодействие с библиотеками MS Windows. В этой главе изучаются классы оконных объектов из библиотеки OWL, и в то же время описывается способ использования этих классов в нашей минимальной программе MIN из библиотеки OWL.

Библиотека OWL предоставляет два оконных класса: TWindow и его предка TWindowObject. Эти классы управляют созданием, уничтожением и организацией окон. Они же подготовят основу для большей части Ваших разработок на базе OWL. На рис.5.1. показано место класса оконных объектов MIN TMinWindow в иерархии классов.

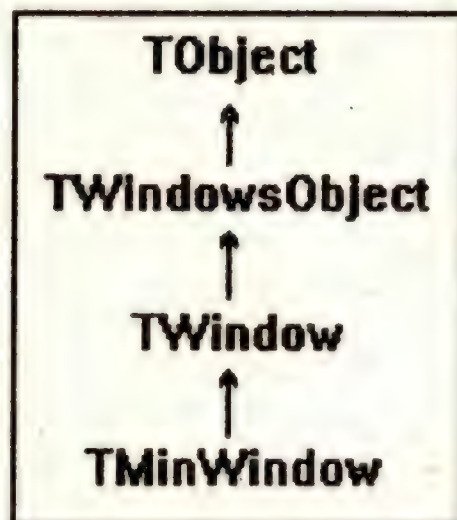


Рис.5.1. Иерархия оконных объектов MIN.

Основной оконный объект из библиотеки OWL содержит функцию, которую вызывают библиотеки Windows: оконную процедуру. Система Windows использует оконные процедуры для передачи Вам информации о Ваших окнах. Библиотека OWL упрощает процесс взаимодействия, позволяя Вам определить набор компонентных функций класса для обработки входящих сообщений. Вместо одной функции, выполняющей всю работу, множество функций - обработчиков сообщений разделяют между собой задачи поддержки окна. Это напоминает бригаду телефонисток, отвечающих на большое количество звонков. Работу может выполнять и одна телефонистка, но, специализируясь на части общей задачи, можно достичь лучших результатов.

Библиотека OWL скрывает от Вас оконные процедуры, а взамен позволяет создавать функции-обработчики сообщений. Задача по написанию кода становится легче при одной функции на сообщение. Библиотека OWL скрывает от Вас эту сложность, однако OWL скрывает и часть того, что заставляет Windows работать. По этой причине мы собираемся углубиться в Windows немного больше, чем Вам может показаться необходимым. Приобретенное понимание позволит Вам с большей легкостью использовать возможности библиотек OWL и Windows.

Рассмотрим вначале класс TWindowObject.

Класс TWindowObject

Система Windows имеет два основных типа окон верхнего уровня: обычные окна прикладной программы и блоки диалога. Основное различие между ними состоит в том, что блоки диалога создаются из шаблона блока диалога. Шаблоны блока диалога являются статическими определениями, созданными во время построения программы и запомненными в качестве ресурса. Диалоги создаются с помощью класса TDialog библиотеки OWL. Вы изучите TDialog в главе 14. Обычное окно прикладной программы, подобно используемому в программе MIN, создается, как потомок класса TWindow библиотеки OWL. Оба класса используют общий базовый класс TWindowObject.

Рассмотрим компоненты данных и компонентные функции, определенные в TWindowObject. Компоненты данных представлены следующим образом:

Тип	Имя	Описание
PUBLIC (Общие):		
HWND	HWindow	Обрабатывает связанное окно Windows
PTWindowObject	Parent	Указывает на порождающий оконный объект - главное окно не имеет предков
int	Status	Флаг ошибки
LPSTR	Title	Копия текста заголовка
PROTECTED (Защищенные):		
FARPROC	DefaultProc	Устанавливаемый по умолчанию обработчик сообщений оконных объектов, который вызывается для всех сообщений Windows, иначе не обрабатываемых
Pvoid	TransferBuffer	Указывает на буфер передачи данных блока диалога
PRIVATE (Частные):		
PTApplication	Application	Указывает на объект TApplication, создавший окно
PTWindowsObject	ChildList	Указывает на порожденное окно, которое находится в головной части списка порожденных элементов - у главного окна MIN нет порожденных
WORD	CreateOrder	Для порожденных окон - целое число, описывающее порядок окон в списке элементов одного уровня
WORD	Flags	Содержит флаги атрибута окна, описывающие такие свойства, как автоматическое создание порожденных окон (для блоков диалога), поддержка многодокументного интерфейса, обработка клавиатуры
FARPROC	Instance	Указатель на блок экземпляра оконного объекта, который является частью механизма диспетчеризации динамических сообщений (не путать с параметром hInstance, передаваемым в WinMain). Подробности изложены в главе 19
PTModule	Module	Указатель на объект модуля
PTWindowsObject	SiblingList	Указатель на следующее окно одного уровня (где окна одного уровня имеют общие порождающие окна)

Класс TWindowsObject содержит 84 компонентные функции: 51 общую, 25 защищенных и 8 частных. Вместо перечисления всех компонентных функций мы представим Вам два коротких списка. Первый список содержит компонентные функции общего доступа. Второй - компонентные функции общего переопределения. Подробности содержатся в документации по библиотеке OWL. Вы можете также обратиться к "первоисточнику" - текстам программ библиотеки OWL.

Компонентные функции общего доступа:

Имя функции	Описание
TWindowsObject()	Конструктор
~TWindowsObject()	Деструктор
CloseWindow()	Закрывает окно после того, как сначала вызывается компонентная функция CanClose, для гарантии того, что закрытие разрешено

GetApplication()	Поиск указателя на объект "прикладная программа", связанный с оконным объектом
GetModule()	Возвращает указатель на объект модуля
SetCaption()	Помещает новый текст в заголовок окна
SetTransferBuffer()	Размещает указатель на буфер для использования при пересылке данных между окнами
Transfer()	Пересылает данные в окно или из окна, такого как блок диалога
TransferData()	Устанавливает флаг направления для компонентной функции Transfer

Компонентные функции общего переопределения:

Имя функции	Описание
CanClose()	Запрашивает, можно ли закрыть окно. Два сообщения Windows WM_CLOSE и WM_QUERYENDSESSION могут вызвать закрытие окна. Но иногда закрыть окно нельзя, например, когда в окне содержатся несохраненные данные. При использовании переопределения этой компонентной функции, можно быть уверенным, что оконный объект сообщит пользователю о любом потенциальном риске потери данных
DefChildProc()	Установленный по умолчанию обработчик уведомляющих сообщений порожденных окон WM_COMMAND
DefCommandProc()	Выполняет установленную по умолчанию обработку сообщения WM_COMMAND
DefWndProc()	Установленный по умолчанию обработчик сообщений для окна. По умолчанию это подпрограмма Windows DefWindowProc. Если Вы переопределяете эту функцию, убедитесь в том, что DefWindowProc вызывается с нежелательными сообщениями
Destroy()	Уничтожает оконный объект и связанный с ним объект пользовательского интерфейса Windows. При переопределении убедитесь в том, что переопределенная функция вызывается последней с той целью, чтобы она смогла выполнить очистку ресурсов
GetClassName()	Используйте для обеспечения уникальных имен для оконных классов Ваших прикладных программ в среде Windows. Никогда не вызывайте переопределенную функцию
GetWindowClass()	Переопределите для модификации значения в структуре данных WNDCLASS для оконного класса Windows. Вызывайте переопределенную функцию первой для инициализации данных класса

Кроме этих компонентных функций, класс TWindowsObject поддерживает специальный тип компонентной функции - обработчик сообщений. Это имеет отношение к функции, которая обрабатывает одиночный тип сообщения. Например, Вы могли бы иметь одну функцию - обработчик сообщений для сообщения WM_PAINT, другую - для сообщения WM_LBUTTONDOWN и третью - для сообщения WM_KEYDOWN. Функции - обработчики сообщений строятся с использованием функционального средства компилятора Borland C++, называемого виртуальной таблицей динамической диспетчеризации (DDVT). Эти таблицы представляют собой массивы, содержащие пары, которые состоят из индекса функции и значений адресов функций.

Диспетчеризация сообщений Windows через DDVT осуществляется очень быстро, быстрее, чем, например, при реализации многострочных операторов switch, типичных для программ в среде Windows, написанных на языке C. Небольшая подпрограмма на языке Ассемблер просматривает таблицу индексов, обращаясь при совпадении к соответствующему обработчику сообщений. Если Вам интересен механизм DDVT, рассмотрите исходные тексты механизма диспетчеризации в DD.ASM.

Для того чтобы определить функцию - обработчик сообщений, Вы обеспечиваете номер индекса определения класса. Например, посмотрим, как программа PIXEL из главы 7 определяет функцию - обработчик сообщений для сообщения WM_PAINT:

```
class TPixelWindow : public TWindow
{
public:
...
virtual void WMPaint(TMessage& Msg) = (WM_FIRST + WM_PAINT);
};
```

Поскольку значение WM_FIRST равно нулю, тот же результат может быть получен из такого определения:

```
virtual void WMPaint(TMessage& Msg) = (WM_PAINT);
```

Как видно из этого текста, оконный класс PIXEL, TPixelWindow строится напрямую не из TWindowsObject, а из TWindow - потомка TWindowObject. Но при этом принцип остается прежним.

Функции - обработчики сообщений могут быть использованы также и другими способами. Например, они могут упростить передачу сообщений от меню и команды кодовых комбинаций клавиш (WM_COMMAND). В частности, вместо одной компонентной функции для каждого меню и кодовой комбинации клавиш Вы можете создать одну функцию-обработчик на команду. Библиотека OWL определяет символическую константу CM_FIRST, которую Вы можете использовать в качестве начального значения для области значений индексов функции WM_COMMAND. Мы рассмотрим это более подробно в главе 11 при обсуждении меню.

Если функция - обработчик сообщений не была создана для определенного сообщения, класс TWindowsObject передает сообщение обработчику по умолчанию. Класс имеет свой собственный обработчик сообщений по умолчанию, DefWndProc(). Но, как и в случае с другими компонентными функциями класса, Вы можете переопределить эту функцию для наблюдения за потоком необработанных сообщений через окно. Конечно, Вы захотите посылать большинство сообщений переопределенной функции. В свою очередь, она будет вызывать обработчик по умолчанию в системе Windows, DefWindowsProc(), который отреагирует на каждое окно в системе аналогичным образом.

Как мы упоминали ранее, RWindowObject является основным классом как для класса блоков диалога библиотеки OWL, так и для класса обычных окон прикладных программ TWindow. Диалоги существенно отличаются от обычных окон прикладных программ. С одной стороны, диалоги обычно создаются из уже существующих оконных классов и включают в себя одиночное всплывающее окно, содержащее группу порожденных окон. Диалог обычно создается из определения ресурса. Диалоги бывают двух разновидностей: модальные и немодальные. Модальный диалог блокирует подачу входных данных своему предку, в то время как немодальный диалог позволяет своему предку продолжать нормально функционировать. И наконец, поскольку блок диалога обеспечивает доступ к прикладным данным, необходим определенный механизм для инициализации значений данных при запуске диалога и для сбора результатов при прекращении диалога.

С другой стороны, окно прикладной программы обычно строится из оконного класса определенной программы. Время его жизни "длиннее", чем время жизни диалога - окно остается на экране до тех пор, пока пользователю нужен доступ к этой программе. Окна прикладной программы никогда не определяются ресурсами, а создаются внутри нее. Хотя окно прикладной программы может иметь порожденные окна, они также должны создаваться внутри прикладной программы, а не считываться из ресурсов.

Мы рассмотрим TDialog в главе 14, когда углубимся в изучение функционирования библиотеки OWL и поддержки блоков диалога Windows. Пока же продолжим изучение программы MIN, исследуя класс TWindow.

Класс TWindow

Поскольку TWindow порождается из класса TWindowsObject, он наследует ряд компонентов данных и компонентных функций. TWindow обрабатывает кое-что из того, что каждая прикладная программа Windows должна выполнить для окна: регистрацию оконного класса MS Windows, прокрутку, перерисование, перемещение и изменение размеров. Класс TWindow также подключается к многодокументному интерфейсу (MDI) поддержки Windows. При построении собственных оконных объектов Вы будете стремиться к тому, чтобы заставить эти основные классы работать на Вас. Давайте посмотрим на компоненты данных TWindow.

Тип	Имя	Описание
PUBLIC (Общие):		
TWindowAttr	Attr	Содержит параметры создания окна, передающиеся при вызове CreateWindowEx: Style, ExStyle, X, Y, W, H, Menu, Id и Param
PTScroller	Scroller	Указывает на объект "механизм прокрутки"
HANDLE	FocusChild Handle	Запоминает логический номер окна, которое имеет фокус ввода с клавиатуры. Это порожденное окно, которому посылаются все сообщения клавиатуры

Класс TWindow имеет небольшое количество компонентных функций - 26, из которых 10 общих, 15 защищенных и 1 частная. Поскольку основная часть Вашей работы с окном будет включать в себя создание функций - обработчиков сообщений, ниже приведены некоторые компонентные функции общего доступа и общего переопределения:

Компонентные функции общего доступа:

Имя функции	Описание
TWindows()	Конструктор
~TWindows()	Деструктор
AssignMenu()	Замещает существующее меню новым меню. Есть две версии этой подпрограммы: одна - для ресурсов меню с ASCII-именами, другая - для ресурсов меню с числовыми идентификаторами ресурса

Компонентные функции общего переопределения:

Имя функции	Описание
Create()	Создает окно MS-Windows, регистрируя при необходимости оконный класс. Эта функция может создавать как окна MDI с документом, так и окна MDI с рамкой. TModule::MakeWindow вызывает для Вас эту функцию, так что Вам никогда не придется прямо вызывать Create. Если Вы переопределяете эту функцию, убедитесь, что вызываете оригинал, поскольку он выполняет для Вас много работы
GetClassName()	Используется для создания уникальных имен оконных классов Ваших прикладных программ в среде Windows. При переопределении не вызывайте переопределенную функцию
GetWindowClass()	Переопределите для модификации значения в структуре данных WNDCLASS для оконного класса Windows. Вызывайте переопределенную функцию первой, поскольку она заполняет эту структуру требуемыми значениями по умолчанию

Для лучшего понимания заранее определенных оконных классов библиотеки OWL рассмотрим программу, которая их использует. По этой же причине обсудим, как наша минимальная программа MIN на базе OWL использует эти классы в своем собственном оконном классе.

Класс TMinWindow программы MIN

Почти каждая программа на базе OWL, которую Вы создаете, будет содержать, по крайней мере, один класс, порожденный классом TWindow. Программа MIN определяет свой класс TMinWindow, который лишь немного отличается от того прочного фундамента, который заложила библиотека OWL. Вот определение класса MIN:

```
class TMinWindow : public TWindow
{
public:
    TMinWindow (PTWindowsObject pwParent, LPSTR lpszTitle,
                PTModule pmModule);
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WNDCLASS&);
};
```

Здесь представлены конструктор и две функции, которые переопределяют порождающий класс. В действительности это определение еще проще, поскольку конструктор пуст. Мы включили его, так как MIN предназначена в качестве шаблона для более сложных программ, которые появятся позже.

Две другие функции класса - GetClassName и GetWindowClass играют свою роль в обеспечении программы MIN частным оконным классом вместе с пиктограммой и курсором. Различные части библиотеки OWL вызывают эти функции для выполнения необходимых действий по созданию для нас окна MS Windows. Для понимания того, что именно вызывается, рассмотрим глубже процесс создания окна.

Создание окон в MS Windows

Создание окна с использованием собственных функций MS Windows представляет собой двух- или трехшаговый процесс. Сначала определяется оконный класс. Затем создается окно. И наконец, окно делается видимым.

Оконный класс немного похож на класс в языке C++. Исполняемый код и данные соединяются в определении класса. Класс служит шаблоном для создания фактических объектов. На этом сходство заканчивается, поскольку классы C++ являются расширяемыми и могут использоваться многократно в отличие от оконных классов. Кроме того, классы C++ обеспечивают больше возможностей с точки зрения инкапсуляции, полиморфизма и наследования.

Оконный класс определяется заполнением структуры WNDCLASS. Этот текст показывает, что типичная Windows-программа делает сама. С другой стороны, регистрация класса незаметно выполняется библиотекой OWL:

```
WNDCLASS wc;
wc.lpszClassName = "MIN:MAIN";
wc.hInstance     = hInstance;
wc.lpfnWndProc   = MinWndProc;
wc.hCursor       = LoadCursor (hInstance, "hand");
wc.hIcon         = LoadIcon (hInstance, "snapshot");
wc.lpszMenuName  = NULL;
wc.hbrBackground = (HBRUSH)COLOR_WINDOW+1;
wc.style         = NULL;
wc.cbClsExtra    = 0;
wc.cbWndExtra    = 0;

RegisterClass( &wndclass);
```

Каждый оконный объект библиотеки OWL связан с одним и только одним оконным классом. Переопределение функции GetClassName позволяет Вам сообщить биб-

лиотеке OWL имя этого класса. Если оно еще не было зарегистрировано, библиотека OWL вызывает функцию `GetWindowClass` для заполнения структуры данных `WNDCLASS`. Библиотека OWL имеет функцию, которая содержит достаточный набор умолчаний. Но когда Вы ее переопределяете, как это сделано в программе `MIN`, то можете изменить любое значение.

Класс `RegisterClass` создает элемент в библиотеке оконных классов `MS Windows`. Во время создания окна функция `CreateWindow` делает ссылку на элемент в этой базе данных. Эта информация становится частью идентификации окна. Рассмотрим все элементы `WNDCLASS`, чтобы лучше понять этот процесс.

У каждого класса есть имя. Вы создаете имя, помещаете его в символьную строку и устанавливаете указатель на строку в элементе `lpszClassName` структуры `WNDCLASS`. Это ключ или справочное значение для элемента базы данных. Функция `GetClassName`, определенная в классе `TMinWindow`, обеспечивает библиотеку OWL именем нашего оконного класса:

```
LPSTR TMinWindow::GetClassName ()
{
    return "MIN:MAIN";
}
```

Если Вы создаете несколько оконных классов в своих программах на базе OWL, убедитесь в том, что каждый класс имеет уникальное имя.

Значение `hInstance` сообщает `Windows` о том, кто создает определение класса. Это необходимо для служебных действий внутри `Windows`. Когда завершается последний экземпляр программы, `Windows` удаляет все связанные определения классов.

Значение `lpfnWndProc` сообщает `Windows` адрес функции для связи с оконным классом. Эта функция называется оконной процедурой. Назначение оконной процедуры формулируется просто: оконная процедура обрабатывает сообщения. В `Windows`-программе, написанной на языке `C`, одна функция обрабатывает множество сообщений, посылаемых окну. В программах на базе OWL, как мы уже обсуждали, Вы определяете различные функции-обработчики сообщений для каждого сообщения, которое хотите обработать. Библиотеки OWL содержат оконную процедуру, поскольку это является требованием `MS Windows`. Но эта процедура следит за вызовом правильной функции - обработчика сообщений исходя из того, как мы определили наш класс оконного объекта.

Значение `hCursor` идентифицирует курсор мыши по умолчанию для оконного класса. При каждом перемещении курсора по нашему окну его форма изменится на ту, которую мы здесь определим. Как уже говорилось ранее, курсор в программе `MIN` был создан с использованием `Resource Workshop`. На файл `MIN.CUR`, содержащий определение курсора, организуется ссылка в нашем файле ресурсов `MIN.RC` в виде строки `hand cursor min.cur`.

Компонентная функция `TMinWindow::GetWindowClass` устанавливает логический номер курсора в `WNDCLASS`. Она вызывает `LoadCursor`, функцию `MS Windows`, которая обеспечивает программу `MIN` требуемым логическим номером курсора:

```
wc.hCursor=LoadCursor(wc.hInstance, "hand");
```

Логический номер присваивается элементу `hCursor` в структуре `WNDCLASS`. Обратите внимание на то, что второй параметр в `LoadCursor` - это имя ресурса `"hand"` в `MIN.RC`.

Элемент `hIcon` в `WNDCLASS` обрабатывается аналогично. Мы создаем пиктограмму, используя `Resource Workshop`, и запоминаем ее в `MIN.ICO`. На это имя файла, в свою очередь, организуется ссылка в файле ресурсов `MIN.RC`, как показано ниже:

```
snapshot icon min.ico
```


Это объявление вызывает связывание данных ресурса пиктограммы (вместе с ресурсом курсора, о котором мы уже говорили) в MIN.EXE во время создания программы. Программа MIN загружает ресурс в память с помощью вызова специальной подпрограммы:

```
wc.hIcon = LoadIcon(hInstance, "snapshot");
```

Определение класса дает возможность определения собственного меню окна. В пределах библиотеки OWL текст, приведенный ниже, устанавливает значение по умолчанию, которое состоит в отсутствии меню класса:

```
wc.lpszMenuName = NULL;
```

Поле hbrBackground в структуре WNDCLASS запрашивает одно, а получает нечто другое. Тут требуется небольшое пояснение. Это поле определяет цвет фона для окна. Если бы нам надо было определить цвет фона для этой напечатанной страницы, мы выбрали бы белый (поскольку книга печатается черными буквами на белой бумаге). Перед тем, как в окне появляется какой-либо рисунок, мы часто закрашиваем окно цветом фона. Это дает нам чистую поверхность для рисования.

Венгерский префикс hbr говорит о том, что это - "ручка кисти" ("handle to a brush"). Мы поговорим о кистях в следующей главе, когда будет представлен интерфейс графических устройств (GDI). Кратко кисть - это способ, с помощью которого программа Windows определяет цвет для заполнения областей. В нашем примере кисть для цвета фона не используется (хотя она могла быть дана).

Вместо этого библиотека OWL вместе с большей частью программ Windows обеспечивает магическое число (логический номер), позволяющее сообщить Windows о том, какой бы мы хотели иметь цвет фона по умолчанию. Пользователь выбирает цвет фона с помощью программы Control Panel. Для того чтобы сделать цвет фона по умолчанию частью определения нашего оконного класса, мы используем особое значение COLOR_WINDOW+1,

```
wc.hbrBackground = COLOR_WINDOW+1;
```

В WINDOWS.H содержится определение

```
#define COLOR_WINDOW 5
```

которое представляет одно из 19 определенных системных констант цвета. Вас могло удивить добавление единицы к этой константе. Дело в том, что первое значение в этой группе равно нулю. А ноль является недействительным значением для логического номера кисти (как и для любого другого логического номера), так что для использования этих констант вместо логического номера мы добавили единицу, чтобы сделать правильным весь диапазон системных цветов.

Следующий элемент структуры WNDCLASS, который мы рассмотрим, - это поле style. В мире программирования в среде Windows термин "style" относится к набору параметров, каждый из которых управляется одним или двумя битами. Для экономии памяти поля style объединяют эти параметры в двух- или четырехбайтовое поле. Мы подробно рассмотрим значение каждого бита style в главе 13, когда будем изучать все аспекты создания окон. По умолчанию библиотека OWL (как и большая часть программ Windows) устанавливает все стили (styles) в положение выключено (off).

```
wc.style = NULL;
```

Последние два поля структуры WNDCLASS - cbClsExtra и cbWndExtra используются для запроса на выделение дополнительного места в базе данных классов (cbClsExtra) и в связанной базе данных - базе данных с информацией об окнах (cbWndExtra), которая содержит запись о каждом окне в системе. Это позволяет Вам присоединить собственные частные данные к классу или окну. Сейчас такое желание может показаться странным. Но подобно тому, как программы на языке C++

инкапсулируют данные в экземпляре класса, окно может инкапсулировать данные об этом экземпляре окна в записи базы данных со сведениями об окнах.

Создание окна

Мы уже говорили о том, что создание окна - это многошаговый процесс. Как только оконный класс определен, следующий шаг состоит в создании окна. Для этого нужна функция `CreateWindowEx`. Вот как эта функция может быть вызвана в версии программы MIN на языке C:

```
hwnd = CreateWindowEx (0L, /* Дополнительный стиль */
    "MIN:MAIN", /* Имя класса */
    "Minimum", /* Заголовок */
    WS_OVERLAPPEDWINDOW, /* Биты стиля */
    CW_USEDEFAULT, /* x - по умолчанию */
    0, /* y - по умолчанию */
    CW_USEDEFAULT, /* cx - по умолчанию */
    0, /* cy - по умолчанию */
    NULL, /* Нет предков */
    NULL, /* Меню класса */
    hInstance, /* Создатель */
    NULL); /* Параметры */
```

В дополнение к `CreateWindowEx` система Windows имеет вторую подпрограмму создания окон - `CreateWindow`. Эти подпрограммы идентичны, отличие лишь в том, что в `CreateWindowEx` - один дополнительный параметр (Ex означает Extended, т.е. расширенный). Этот дополнительный параметр стоит первым в списке параметров, но не потому, что он самый важный, а просто потому, что так легче вызвать новую подпрограмму вместо старой. Рассмотрим параметры этих подпрограмм.

Первый параметр, имеющийся только в `CreateWindowEx`, содержит биты стиля. Как указывалось выше, поля стиля объединяют несколько одно- или двухбайтовых полей в одно-, двух- или четырехбайтовое значение. Этот дополнительный параметр был нужен, так как, откровенно говоря, при вызове старой `CreateWindow` происходил выход за пределы памяти в поле стиля. Первоначальное поле стиля является третьим параметром при вызове `CreateWindow` и четвертым параметром `CreateWindowEx`. Библиотека OWL устанавливает его в 0L, так как типичное главное окно не требует никакого стиля.

Второй параметр - `CreateWindow` — определяет класс окна, которое мы хотим создать. В нашем примере имя класса - это MIN:MAIN.

Третий параметр - это заголовок окна. Заголовок окна появляется в окне при его открытии и рядом с пиктограммой окна при его минимизации. В заголовке удобно держать имя нашей программы. После создания окна Ваша программа может изменить заголовок, вызвав подпрограмму `SetWindowText`. В классе `TMinWindow` заголовок окна определен как второй параметр конструктора:

```
MainWindow = new TMinWindow (NULL, "Minimum", NULL);
```

Четвертый параметр предназначен для задания битов стиля. Библиотека OWL использует стандартное определение `WS_OVERLAPPEDWINDOW` для главного окна прикладной программы. Это определение объединяет вместе шесть простых значений стиля:

```
#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED |
    WS_CAPTION | WS_SYSMENU | WS_THICKFRAME |
    WS_MINIMIZEBOX | WS_MAXIMIZEBOX)
```

Логический оператор ИЛИ в языке C упаковывает вместе эти битовые определения:


```
#define WS_OVERLAPPED 0x00000000L
#define WS_CAPTION 0x00C00000L
#define WS_SYSMENU 0x00080000L
#define WS_THICKFRAME 0x00040000L
#define WS_MINIMIZEBOX 0x00020000L
#define WS_MAXIMIZEBOX 0x00010000L
```

В определении `WS_OVERLAPPED` нет битов со значением "1". Это поле игнорируется при логическом сложении с другими битовыми значениями. Однако этот флаг стиля может быть использован для того, чтобы различить три типа окна, которые могут быть созданы. Два других типа создаются с использованием следующих флагов стиля:

```
#define WS_POPUP 0x80000000L
#define WS_CHILD 0x40000000L
```

Таким образом, `WS_OVERLAPPED` - это просто метка-заполнитель, благодаря которой Вы с первого взгляда видите, окно какого типа создается.

Другие биты стиля, которые являются частью определения `WS_OVERLAPPED`, служат для создания различных частей окна. Полное описание всех различных битов стиля окна будет приведено в главе 13.

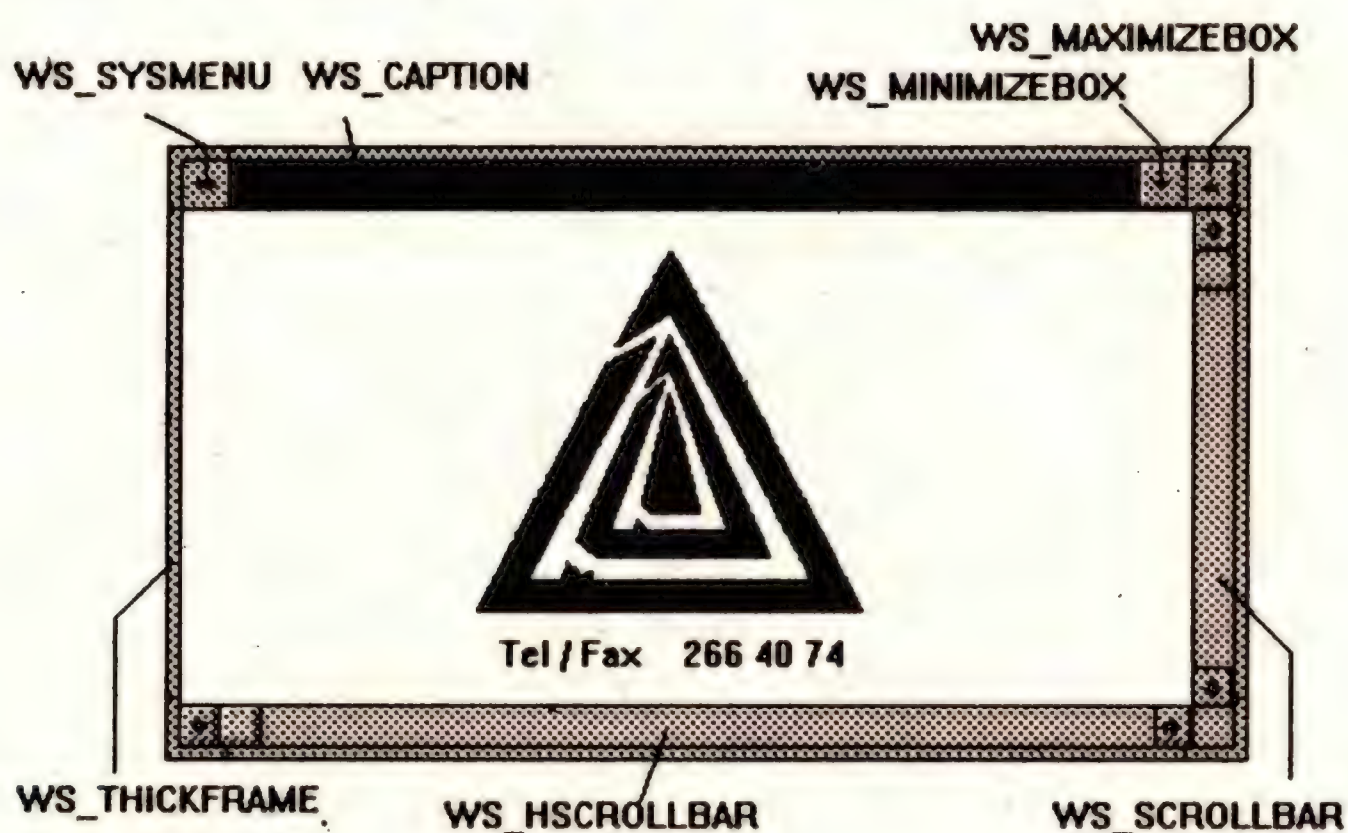


Рис.5.2. Биты определения стиля.

Пятый, шестой, седьмой и восьмой параметры используются для того, чтобы определить начальный размер и положение нашего окна. Значения `X` и `Y` описывают положение верхнего левого угла нашего окна по отношению к верхнему левому углу экрана дисплея. Единицей измерения для всех четырех параметров служит пиксель. Библиотека OWL так же, как и большая часть программ Windows, находит простой выход в запросе Windows на создание нашего окна с подходящими размером и положением с помощью флага `CW_DEFAULT`. Этот флаг должен быть установлен в полях `X` и `CX` функции `CreateWindow`. Когда он используется, значения `Y` и `CY` игнорируются.

Девятый параметр определяет порождающее окно. Хотя термин "родитель" имеет множество дополнительных значений как в реальном мире, так и в мире объектно-ориентированного программирования, суть его здесь довольно проста. Он определяет, где существует окно (для окна `WS_CHILD`) и может ли окно быть автоматически показано/скрыто/уничтожено (для всех типов окон). Когда окно показывается, скрывается или уничтожается, все порожденные им окна разделяют его судьбу. Подобно главному окну в большинстве программ, библиотека OWL создает главное окно программы MIN без порождающего окна.

Десятый параметр позволяет присоединить меню к окну. Это наша вторая возможность для определения окна (первая была при определении класса). Структура данных WNDCLASS позволяет определить меню, которое бы совместно использовалось всеми элементами класса. Если бы наше определение класса содержало имя меню, мы бы автоматически получили меню класса, задав NULL в этом параметре. Если окно имеет собственное меню, оно может использовать для этой цели поле. По умолчанию библиотека OWL определяет главное окно MIN без меню, а также устанавливает девятый параметр функции CreateWindow в NULL.

Одиннадцатый параметр - логический номер экземпляр - идентифицирует владельца окна, что позволяет Windows знать, какой именно экземпляр программы создал окно, в результате система Windows получает возможность правильно установить регистр сегмента данных при вызове оконной процедуры.

Двенадцатый параметр позволяет передать указатель на данные в оконную процедуру. Указатель передается с самым первым сообщением WM_CREATE, что необходимо для обеспечения данных для инициализации окна.

Функция CreateWindowEx возвращает идентификатор созданного окна - логический номер окна. После возврата из CreateWindowEx система Windows создает всю внутреннюю информацию, необходимую для сопровождения окна, т.е. вносит запись в базу данных, содержащую информацию об окнах. Но наше окно на экране еще не появилось - требуется вызвать ShowWindow. При выводе окна программа использует стиль пользователя. Подобно большинству прикладных программ Windows, программы библиотеки OWL используют этот параметр в ShowWindow примерно так:

```
ShowWindow(hwnd, nCmdShow);
```

В зависимости от значения nCmdShow программа может при вызове иметь минимизированное (до размеров пиктограммы), обычное или максимизированное (до размеров всего экрана) окно. Вызовом ShowWindow завершается инициализация оконного объекта. Такая последовательность действий при инициализации типична для программы Windows. Перед тем как мы продолжим изучение интерфейса прикладных программ MS Windows, исследуем, как программа из библиотеки OWL может лучше настроить процесс создания окна.

Создание окна и библиотека OWL

Вы уже видели, как программа из библиотеки OWL может управлять регистрацией оконного класса. Она может переопределить функцию GetWindowClass, чтобы изменить значения в структуре WNDCLASS. Вот так MIN добавила курсор и пиктограмму пользователя к определению класса.

Переопределение параметров в CreateWindowEx - достаточно простой процесс. Вспомните, что один из компонентов данных RWindow - это Attr. Он является структурой типа TWindowAttr. Если Вы посмотрите исходные тексты библиотеки OWL, то увидите, что эта структура определена как

```
struct _CLASSTYPE TWindowAttr {  
    DWORD Style;  
    DWORD ExStyle;  
    int X, Y, W, H;  
    LPSTR Menu; // Имя меню  
    int Id; // Идентификатор порожденного окна  
    LPSTR Param;  
};
```

Это параметры, которые используются при вызове CreateWindowEx.

Конструктор TWindow заполняет эту структуру набором значений по умолчанию. Поскольку конструктор производного класса вызывается после конструктора основного класса, Вы можете изменить любое значение в Вашем конструкторе, и оно будет использовано при вызове на создание окна. Например, программа MIN создаст окно

размером 200 пикселей в высоту и 300 пикселей в ширину при изменении конструктора TMinWindow на

```
TMinWindow::TMinWindow (PTWindowsObject pwParent,  
    LPSTR lpszTitle, PTModule pmModule)  
:TWindow (pwParent, lpszTitle, pmModule)  
{  
    Attr.H = 200;  
    Attr.W = 300;  
}
```

Изменяя остальные значения в компоненте данных Attr, Вы можете изменить способ создания Ваших окон.

Одно из значений, передаваемых в подпрограмму RegisterClass, является адресом функции, известной как оконная процедура. Когда Ваша программа использует библиотеку OWL, Вы можете не определять этот тип функции, потому что библиотека OWL предоставит его Вам сама. Оконная процедура библиотеки OWL вызывает компонентную функцию Вашего оконного объекта. Однако, поскольку оконные процедуры представляют собой центральную часть программирования в среде Windows, перейдем к их описанию.

Объявление оконной процедуры

Каждая оконная процедура, находится ли она в библиотеке OWL или в программе для Windows, написанной на языке C, должна быть определена следующим образом:

```
long FAR PASCAL MinWndProc (HWND hwnd,  
    WORD msg,  
    WORD wParam,  
    LONG lParam)
```

Имя процедуры, как и имя каждого параметра, можно определять по собственному усмотрению. Но количество и тип параметров процедуры должны соответствовать приведенному описанию. Вам не нужно писать программный код для вызова этой подпрограммы, оконная процедура является частью программы для Windows с пассивным ожиданием сообщений, т.е. эта подпрограмма вызывается системой Windows напрямую.

Значение возврата из оконной процедуры имеет тип long. Тип значения возврата зависит от сообщения. Например, сообщение WM_QUERYENDSESSION ожидает значение TRUE (ненулевое) или FALSE (нулевое). Именно это сообщение позволяет программе голосовать за выполнение Windows требования пользователя закрыть систему (Windows может быть очень демократичной системой). Сообщение WM_GETTEXT, с другой стороны, ожидает получения дальнего указателя на строку (char far *) в качестве значения возврата.

Каждая оконная процедура имеет ровно четыре параметра: логический номер окна (hwnd), идентификатор сообщения (msg) и два значения параметров (wParam и lParam).

Логический номер окна hwnd идентифицирует окно, связанное с сообщением. Этот параметр необходим, так как одна оконная процедура может поддерживать несколько различных окон одновременно. Конечно, он не представляет большого интереса в программе из библиотеки OWL, поскольку оконные объекты уже содержат значение логического номера окна в своем компоненте данных HWindow.

Второй параметр msg является оконным сообщением. Если логический номер окна несет информацию о том, "кто вызывает", то значение сообщения говорит нам, "что они хотят". Например, можно сообщить о том, что окно было порождено (WM_CREATE), находится в новом положении (WM_MOVE), увеличилось (или

уменьшилось) в размерах (WM_SIZE), нуждается в изменении внешнего вида (WM_PAINT) или исчезло (WM_DESTROY).

Параметры wParam и lParam предоставляют дополнительную информацию о данном сообщении. Параметр wParam определен как WORD, который в WINDOWS.H задан 16-битовым целым без знака. Параметр lParam определен как LONG, который является 32-битовым значением без знака. Значения wParam и lParam зависят от типа сообщения. Например, сообщение WM_COMMAND, которое посылается при выборе пункта меню, использует wParam для идентификации того пункта меню, который был выбран. Сообщения от мыши, такие как WM_LBUTTONDOWN и WM_MOUSEMOVE, используют lParam для запоминания положения курсора мыши в "клиентной" области окна.

Ключевое слово PASCAL заставляет функцию использовать соглашение о связях, принятое в Паскале. Это ключевое слово не является частью стандарта C, предложенного ANSI, но представляет собой расширение языка C. Этой уловкой воспользовались в архитектуре Intel для получения более компактного и быстрого кода, чем при использовании соглашения о связях, принятого по умолчанию компилятором языка C. В действительности все подпрограммы интерфейса прикладных программ Windows (за исключением wsprintf) используют это соглашение о связях.

Ключевое слово FAR (дальний) необходимо, потому что оконная процедура всегда вызывается с использованием межсегментного вызова. Родственное ключевое слово NEAR (ближний) определяет ссылку на то, что находится в том же сегменте. Программисты часто прикладывают много усилий для уменьшения количества дальних ссылок из-за больших издержек по сравнению с ближними ссылками. Подробности - в приложении H.

Оконные объекты из библиотеки OWL обеспечивают оконную процедуру, так что Вам не требуется создавать свою собственную. Понимание оконной процедуры поможет ориентироваться в текстах для Windows, написанных на языке C. И, конечно, это поможет Вам оценить, как библиотека OWL скрывает от Вас некоторую сложность системы Windows.

Хотя Вам никогда не придется создавать оконную процедуру, Вы, конечно, создадите функции - обработчики сообщений. Как уже говорилось, это функции, которые Вы определите как часть оконных объектов для обработки конкретных сообщений. Давайте рассмотрим, что требуется для типичной функции - обработчика сообщений.

Функции - обработчики сообщений из библиотеки OWL

Для каждого сообщения, которое будет обрабатывать оконный объект, Вы создадите функцию - обработчик сообщений. Как Вы, наверное, помните, функция - обработчик сообщений является компонентной функцией класса оконного объекта. В некотором смысле каждая функция является крошечной оконной процедурой. В конце концов, оконная процедура библиотеки OWL - это всего лишь "телефонистка", передающая сообщения функции - обработчику сообщений.

Функция - обработчик сообщений отличается от оконной процедуры в объявлении. Они обе обеспечивают доступ к одним и тем же четырем параметрам, которые несколько отличны по упаковке. Например, вот как функция - обработчик сообщений определена в LINES - одном из примеров интерфейса графических устройств (в главе 8):

```
void TLinesWindow::WMLButtonDown( TMessage& Msg )
```

Упаковка параметров требует объяснений. Прежде всего, StdWndProc - оконная процедура из библиотеки OWL - копирует параметры оконной процедуры в структуру типа TMessage, определенную в WINDOBJ.H как:

```
struct TMessage {  
    HWND Receiver;  
    WORD Message;  
    union {
```



```

    WORD WParam;
    struct tagWP {
        BYTE Lo;
        BYTE Hi;
    } WP;
};
union {
    DWORD LParam;
    struct tagLP {
        WORD Lo;
        WORD Hi;
    } LP;
};
long Result;
};

```

По существу здесь представлен стандартный набор параметров оконных процедур: Receiver, Message, WParam и LParam. Пятый параметр структуры Result служит для запоминания возвращаемого значения. Тип значения возврата зависит от типа сообщения. Например, сообщение WM_GETTEXT ожидает дальний указатель на символьную строку.

Если Вы новичок в программировании на языке C++, то использование & в параметре WMLButtonDown может показаться не совсем обычным. Это параметр, передаваемый по ссылке, т.е. параметр, обеспечивающий доступ к фактической ячейке памяти, используемой вызывающей программой. Данный подход отличается от подхода "передача по значению", к которому привыкли программисты, пишущие на языке C. Последний метод передачи параметров заключается в использовании временных переменных для запоминания значений параметров.

Параметр, передаваемый по ссылке очень похож на передаваемый указатель. А передача указателей функциям знакома каждому программисту, работающему на языке C. Например, стандартная функция memcpy получает два указателя:

```

char * pchInBuffer;
char * pchOutBuffer;
int cbBuffer;

```

```

memcpy (pchOutBuffer, pchInBuffer, cbBuffer)

```

Параметр, передаваемый по ссылке, немного отличается от указателя. В частности, вызываемая функция вместо оператора ">" использует оператор "." так, будто структура была объявлена самой вызываемой функцией, в результате функция - обработчик сообщений, подобная WMLButtonDown, получает доступ к компонентам TMessage. Вот, например, как можно получить доступ ко всем пяти компонентам TMessage:

```

HWND    hwnd;
WORD    wMessage;
WORD    wParam;
LONG    lParam;
hwnd     = TMessage.Receiver
wMessage = TMessage.Message
wParam   = TMessage.WParam
lParam   = TMessage.LParam

```

```

TMessage.Result = 0L;

```

Этот фрагмент кода подчеркивает различие между обычной Венгерской Записью и соглашением, принятым в библиотеке OWL. Мы решили, что это небольшое отк-

лонение, характерное для библиотеки OWL, может оказаться полезным для того, чтобы отличать переменные библиотеки OWL от Ваших собственных.

Внутри функции - обработчика сообщений есть два способа доступа к логическому номеру окна. Во-первых, он запоминается в компоненте данных HWindow класса TWindowsObject, а кроме того, хранится в поле Receiver структуры TMessage. С точки зрения эффективности одинаковые машинные команды используются для доступа к каждой из двух переменных. Выбор зависит от того, что Вы сами предпочитаете. Вообще, мы стремились выбрать компонент данных HWindow.

Время от времени Вам будет нужен доступ к части параметра сообщения. Например, сообщения от мыши объединяют положения курсора по осям X и Y в параметре LParam (Y - в старшем слове, X - в младшем). Несмотря на то что есть макропроцедуры, позволяющие разделить эти значения, определение TMessage делает это удобным. Вот как можно получить доступ к старшему слову параметра LParam:

```
WORD wHigh;  
wHigh = Msg.LP.Hi
```

Основное преимущество использования параметра, передаваемого по ссылке в функциях - обработчиках сообщений, - это скорость. В конце концов, передача указателя (4 байта) требует меньших издержек, чем передача всей структуры TMessage (14 байтов). При передаче большого количества сообщений через систему экономия будет существенной.

Каждая функция - обработчик сообщений будет определяться так, как в нашем примере, поэтому Вы можете использовать ее в качестве шаблона для каждой программы из библиотеки OWL, которую пишете. Этот шаблон мы использовали для всех примеров программ в этой книге. Стоит упомянуть об одном небольшом отклонении - использовании RTMessage вместо TMessage&:

```
void TLinesWindow::WMLButtonDown( RTMessage Msg ) .
```

Примеры такого использования Вы найдете в демонстрационных программах из библиотеки OWL, которые фирма Borland поставляет вместе с библиотекой OWL. Оба выражения эквивалентны. Вы можете выбрать то, которое Вам больше нравится.

На данном этапе Вы уже имеете представление об основах использования оконных объектов из библиотеки OWL для создания программ под Windows. Однако есть существенный момент, который мы еще не затрагивали: правильное завершение программы. Хотя сначала может показаться, что это работа объекта "прикладная программа", в действительности завершение программы в системе Windows обычно связано с жизнью главного окна программы. Давайте рассмотрим, как оконный объект из библиотеки OWL обрабатывает завершение программы.

Завершение программы

Для каждой программы под Windows важным является вопрос правильного завершения. Правильное завершение необходимо для очистки (освобождения) ресурсов и возврата памяти и других ресурсов, используемых программой. При обсуждении объекта "прикладная программа" мы ввели сообщение WM_QUIT. Когда появляется сообщение WM_QUIT, цикл обработки сообщений завершается. Давайте посмотрим, как сообщение WM_QUIT попадает в очередь сообщений.

Завершение программы обычно происходит при закрытии главного окна программы. Пользователь может закрыть окно различными способами, в частности, выбрать функцию Close в системном меню или нажать комбинацию клавиш Alt + F4. Ниже приведен список сообщений, которые направляются оконной процедуре при закрытии окна программы MIN с помощью комбинации клавиш Alt+F4:

Сообщение	Комментарий
WM_SYSKEYDOWN	Была нажата клавиша Alt
WM_SYSKEYDOWN	Была нажата клавиша F4
WM_SYSCOMMAND	Сгенерирована системная команда
WM_CLOSE	Окну приказано закрыться
WM_NCACTIVATE	Выключена подсветка заголовка окна (неклиентная область)
WM_ACTIVATE	Окно становится неактивным
WM_ACTIVATEAPP	Прикладная программа становится неактивной
WM_KILLFOCUS	Окно "отсоединяется" от клавиатуры
WM_DESTROY	Окно уничтожено
WM_NCDESTROY	Пора очистить "неклиентную" область данных

Первое, что можно сказать об этом потоке сообщений, - то, что оконная процедура, устанавливаемая по умолчанию, выполняет основную часть работы по обслуживанию окна. Например, она принимает первые два сообщения, представляющие собой необработанные сообщения о нажатии клавиш, и переводит их в третье сообщение - о системной команде, которое обрабатывается оконной процедурой, устанавливаемой по умолчанию для выполнения команд системного уровня.

Системная команда представляет собой групповой запрос на выполнение некоторого действия (переместить окно, изменить размер окна и т.д.). Оконный объект обычно не обрабатывает эти сообщения - он следит за наиболее специфическими сообщениями, на которые нужно ответить. В нашем случае таким сообщением является WM_CLOSE - запрос на закрытие нашего окна.

Класс TWindowsObject имеет компонентную функцию WMClose для обработки этого сообщения, которая сначала выясняет, можно ли закрыть окно. Для этого она вызывает одну из двух различных компонентных функций с именем CanClose. Если рассматривается главное окно прикладной программы, то вызывается функция CanClose объекта "прикладная программа", если любое другое окно - функция CanClose оконного объекта. В обоих случаях для объекта "прикладная программа" и для оконного объекта эта компонентная функция возвращает одно из двух значений: TRUE (ненулевое), позволяющее окну закрыться, или FALSE (нулевое), препятствующее закрытию окна.

По умолчанию действие TApplication::CanClose состоит в вызове функции CanClose главного окна. По этой причине мы решили, что компонентную функцию можно рассматривать только в отношении реализации оконного объекта. Версия объекта "прикладная программа" облегчает запрос о закрытии прикладной программы из любой ее части.

Типичная компонентная функция CanClose могла бы проверять наличие несохраненных файлов в нашем окне. Если есть такой файл, то наш оконный объект мог бы показать на экране блок вывода сообщений с экранными кнопками "Yes", "No" и "Cancel" для предупреждения пользователя о возможной потере данных (рис.5.3). Компонентная функция CanClose убеждается в том, что закрытие окна не приведет к нежелательным последствиям. В примере блока вывода сообщений функция CanClose возвратит сообщение TRUE (можно закрывать), если пользователь нажал кнопку "Yes" или "No", если "Cancel", то CanClose вернет FALSE для предотвращения закрытия окна.

Класс TWindowsObject обрабатывает сообщение WM_QUERYENDSESSION в своей компонентной функции WMQueryEndSession. Система Windows отправляет это сообщение главному окну каждой программы, когда пользователь собирается закрыть систему. По существу, голосование закончено. Для завершения Windows необходим единогласный ответ. Повторим: CanClose вызывается для того, чтобы получить гарантию от потери данных пользователя. Вначале вызывается функция

CanClose прикладной программы, которая, в свою очередь, вызывает компонентную функцию CanClose главного окна.

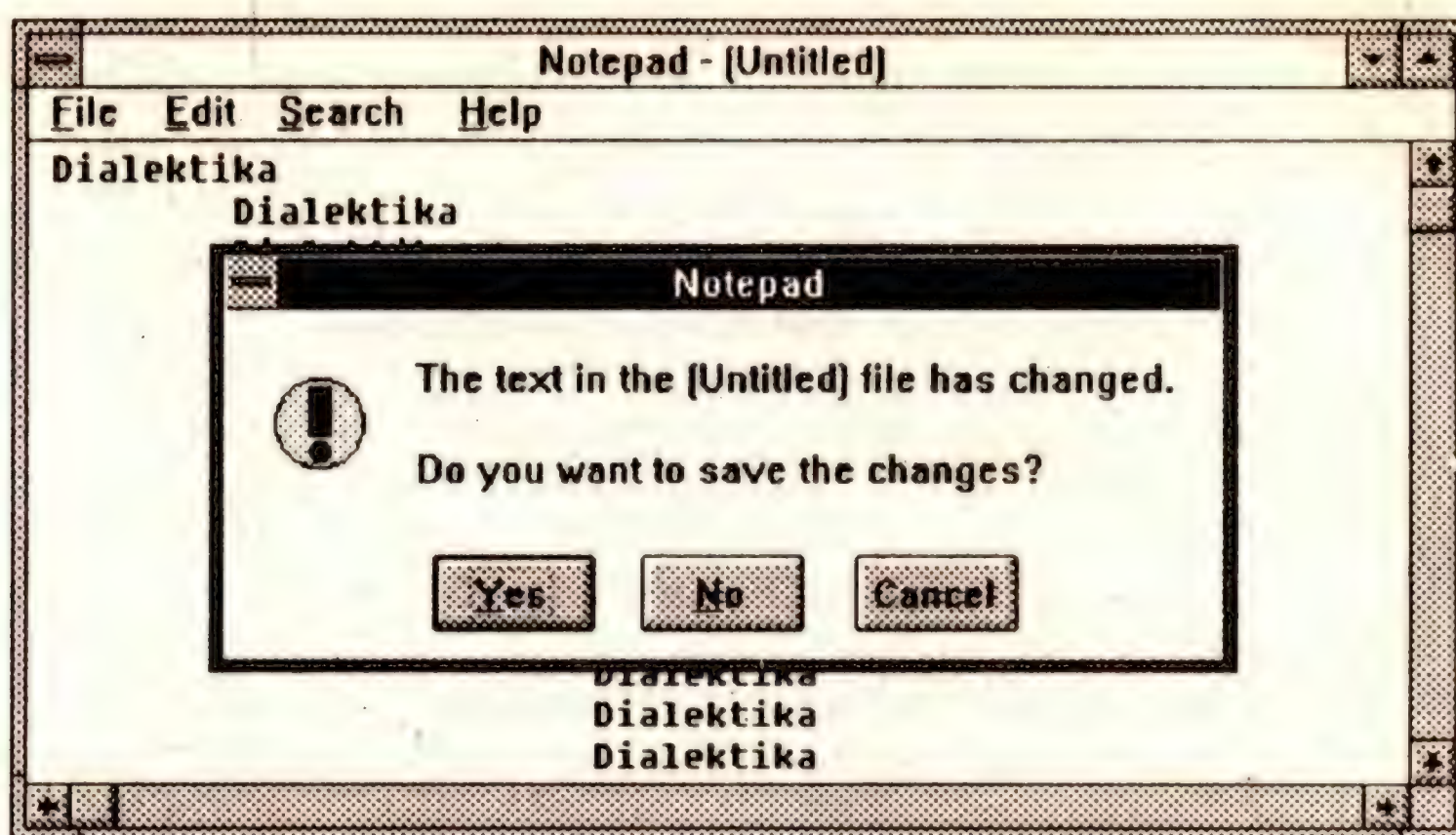


Рис.5.3. Защита от потери данных.

Когда функция CanClose сообщает, что можно закрывать окно, то вызывается компонентная функция Destroy. Это компонентная функция TWindowsObject, очищающая окна MDI с документами, которые были созданы. Затем вызывается процедура DestroyWindow системы MS Windows, полностью и безвозвратно уничтожающая окно. Она отправляет информацию об этом в форме сообщения WM_DESTROY.

Итак, класс TWindowsObject имеет компонентную функцию для обработки этого важного сообщения. Если есть главное окно прикладной программы, вызывается подпрограмма PostQuitMessage системы Windows, которая ставит сообщение WM_QUIT в очередь сообщений прикладной программы. Оно, как Вы помните, ответственно за завершение цикла ожидания сообщений прикладной программы, так что программа сама может завершиться.

Как Вы убедились, большую часть работы по завершению программы берут на себя библиотеки OWL. Конечно, мы можем вмешаться в этот процесс путем переопределения компонентных функций основного класса. Важную роль в обработке сообщений играет оконная процедура системы Windows, устанавливаемая по умолчанию. Давайте уделим внимание этой "рабочей лошадке".

Обработка сообщений, устанавливаемая по умолчанию

Как было сказано в главе 1, главной причиной притягательности Windows является последовательный принцип интерфейса пользователя Windows "смотри и воспринимай". Даже непостоянные пользователи Windows отмечают замечательное единообразие программ. Например, меню действуют одинаково между двумя программами, в других отношениях не связанных между собой, и окна в различных программах отвечают на одни и те же действия совершенно одинаково.

Углубляясь в систему Windows, можно сказать, что это работа обработчика окон, устанавливаемого по умолчанию и отвечающего за высокую степень единообразия. Поток сообщений, который оконный объект отводит к обработчику, устанавливаемому по умолчанию, и есть причина сходства между программами. В то время как программа, написанная на языке C, будет полагаться на оконную процедуру, устанавливаемую по умолчанию, программа из библиотеки OWL воспользуется сервисом основных объектов из библиотеки OWL. Но даже программа из библиотеки OWL полагается до некоторой степени на работу, выполненную оконной процедурой, которая устанавливается по умолчанию.

Оконный объект играет роль полицейского, регулирующего уличное движение для сообщений. В конце концов, обработчик сообщений, устанавливаемый по умолчанию, никогда не вызывается прямо системой Windows. Вместо этого оконные объекты всегда первыми получают сообщения. Для того чтобы интерфейс пользователя работал правильно, библиотека OWL передает сообщения оконной процедуре, устанавливаемой по умолчанию.

Может возникнуть необходимость изменить поведение окна по сравнению с поведением по умолчанию. Например, мы захотели бы воспрепятствовать пользователю захватить окно за заголовок. (Для пользователя это типичный способ переместить окно.) Простой способ состоит в отлавливании сообщений, связанных с этим действием для того, чтобы они не поступали обработчику окон, устанавливаемому по умолчанию. Для большей же части, однако, мы позволим оконной процедуре, устанавливаемой по умолчанию, выполнять для нас свою работу, так что наша программа может принимать участие в последовательности, которая делает систему Windows такой удобной.

Рассматривая различные части минимальной программы для Windows, мы много говорили о потоке сообщений по системе. Мы также идентифицировали несколько важных сообщений. Для того чтобы Вы хорошо поняли, как может работать конвейер сообщений с пользой, Вам было бы полезно узнать типы сообщений, проходящих по этому конвейеру. Поэтому мы хотим более подробно остановиться на типах сообщений в системе Windows.

Систематика сообщений

В системе Windows имеется около 250 стандартных сообщений. К счастью, Вам не нужно знакомиться со всеми, поскольку большинство из них используются очень редко. Например, большое количество сообщений характерно для одного типа окна. Другие сообщения играют особую роль в совместном использовании данных или реализации стандарта многодокументного интерфейса (MDI). Третьи - ходят по нашему конвейеру сообщений, но не являются документами для пользования. Это внутренние сообщения, которые система Windows создает для своих собственных целей.

Чтобы прийти к соглашению о сообщениях, мы разделим их на восемь категорий. На этом изучение сообщений не закончится. Почти в каждой главе нашей книги Вы найдете подробное описание сообщений, которые относятся к различным частям системы Windows. Начнем с рассмотрения сообщений системы Windows.

Сообщения имеют имена. Они определены в `WINDOWS.H` примерно так:

```
#define WM_COMMAND 0x0111
```

Венгерский префикс `WM_` говорит о том, что символ является "оконным сообщением". В устной речи префикс игнорируется, так что программисты часто ссылаются на `WM_COMMAND` как на сообщение "команда". Эти символические имена помогают задать числовые значения сообщений в 16-битовом целом без знака. Кроме префикса `WM_`, есть и другие префиксы для сообщений: `EM_`, `BM_`, `DM_`, `LB_` и `SB_`. Это частные сообщения, которые пока нас не интересуют, правда, они показывают, что Вы можете создавать собственные частные сообщения по мере надобности.

Утилита WinSight является полезным инструментом для изучения сообщений. Фирма Borland предоставляет эту утилиту вместе с версией 3.x компилятора Borland C++. Она поможет перехватить поток сообщений для окна, даже если Вы не написали программу. При использовании WinSight Вы будете поражены большим количеством обнаруженных сообщений. Если Вы проведете эксперимент с WinSight, то увидите, что такой поток сообщений типичен при любых действиях пользователя.

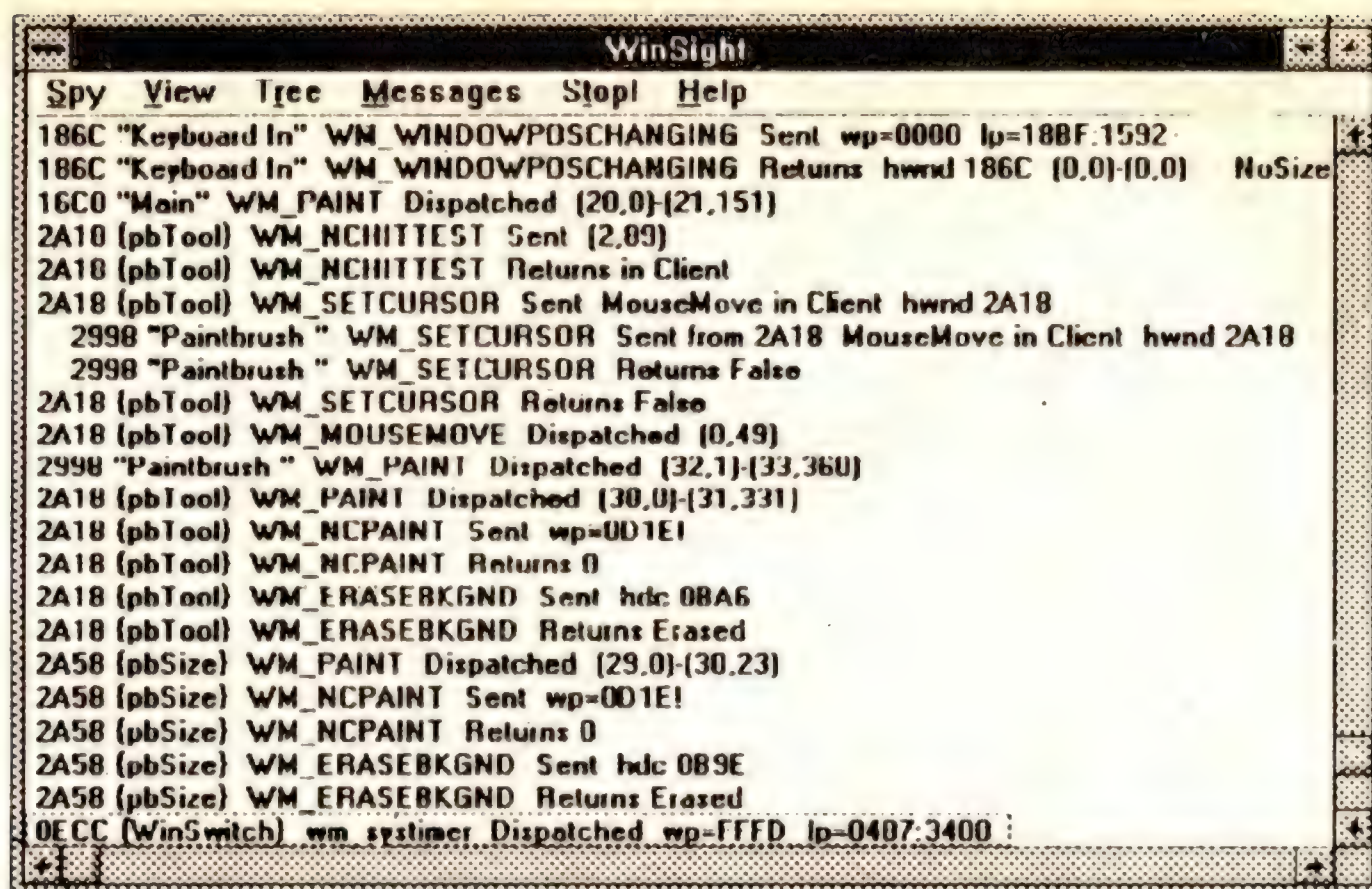


Рис.5.4. Вывод на экран сообщений с помощью утилиты WinSight.

Почему так много сообщений? Некоторые из них предназначены для внутреннего использования в системе Windows, для синхронизации событий в различных частях системы. Они передаются оконной процедуре, устанавливаемой по умолчанию, о которой мы недавно говорили. Остальные сообщения уведомляют оконную процедуру о действиях, предпринятых пользователем. Сообщения гарантируют, что оконная процедура будет всегда иметь последнюю информацию о действиях пользователя. Как только Вы выяснили тип генерируемых сообщений, с Вашей стороны потребуется меньше усилий по поддержанию связи с этой информацией.

В табл.5.1 представлены восемь категорий сообщений. Когда мы будем делать обзор каждой категории, мы перечислим относящиеся к ней сообщения. Поскольку некоторые из категорий сильно специализированы, мы ограничим обсуждение кратким введением и пропустим список сообщений. Если же Вам интересно, что именно мы пропустили, то в приложении А представлен список сообщений всех категорий. Давайте начнем с первой категории - аппаратных сообщений.

Таблица 5.1. Восемь типов сообщений.

Тип сообщения	Описание
Аппаратные	Входные данные от мыши и с клавиатуры
Об организации окна	Уведомление, требование действия, запрос
Об организации интерфейса пользователя	Меню, указатель мыши, линейка прокрутки, блоки диалога, MDI
О завершении	Заккрытие прикладной программы или системы
Частные	Элементы управления блоком диалога: редактор, кнопка, блок списка, комбо-блок
С уведомлением о системном ресурсе	Изменение цвета, шрифты, буферизация печати, режимы работы устройства
О совместном использовании данных	Буфер вырезанного изображения и динамический обмен данными (DDE)
Внутренние системные	Недокументированные сообщения

Аппаратные сообщения

Оконная процедура получает сообщения, генерируемые тремя различными частями аппаратного обеспечения: клавиатурой, мышью и системным таймером. Все они гене-

рируют аппаратные прерывания. Поскольку диспетчеризация Windows не действует в режиме прерываний, аппаратные события должны быть буферизированы. Это обеспечивает обработку событий в порядке их поступления. Например, когда Вы нажимаете клавишу "H", прерывание уведомляет систему о том, что приготовлены данные с клавиатуры. Драйвер устройства клавиатуры Windows отыскивает эти данные и создает элемент в очереди аппаратных событий. Когда наступает его очередь, сообщения, содержащие информацию о клавише "H", направляются в соответствующую оконную процедуру. Сообщения от мыши и системного таймера обрабатываются аналогично. В табл.5.2 представлены 29 сообщений, которые генерируются в ответ на работу аппаратного обеспечения.

Таблица 5.2. Сообщения от аппаратуры.

Сообщения от мыши в "клиентной" области окна:

WM_LBUTTONDOWNBLCLK	Двойное нажатие левой кнопки
WM_LBUTTONDOWN	Нажатие левой кнопки
WM_LBUTTONUP	Отпускание левой кнопки
WM_MBUTTONDOWNBLCLK	Двойное нажатие средней кнопки
WM_MBUTTONDOWN	Нажатие средней кнопки
WM_MBUTTONUP	Отпускание средней кнопки
WM_MOUSEMOVE	Перемещение мыши
WM_RBUTTONDOWNBLCLK	Двойное нажатие правой кнопки
WM_RBUTTONDOWN	Нажатие правой кнопки
WM_RBUTTONUP	Отпускание правой кнопки

Сообщения от мыши в "неклиентной" области окна:

WM_NCLBUTTONDOWNBLCLK	Двойное нажатие левой кнопки
WM_NCLBUTTONDOWN	Нажатие левой кнопки
WM_NCLBUTTONUP	Отпускание левой кнопки
WM_NCMBBUTTONDOWNBLCLK	Двойное нажатие средней кнопки
WM_NCMBBUTTONDOWN	Нажатие средней кнопки
WM_NCMBUTTONUP	Отпускание средней кнопки
WM_NCMOUSEMOVE	Перемещение мыши
WM_NCRBUTTONDOWNBLCLK	Двойное нажатие правой кнопки
WM_NCRBUTTONDOWN	Нажатие правой кнопки
WM_NCRBUTTONUP	Отпускание правой кнопки

Сообщения от клавиатуры:

WM_CHAR	Ввод символа
WM_DEADCHAR	Диакритический знак (умляут, знак ударения и т.д.)
WM_KEYDOWN	Была нажата клавиша
WM_KEYUP	Была отпущена клавиша
WM_SYSCHAR	Ввод специального символа
WM_SYSDEADCHAR	Специальный диакритический знак
WM_SYSKEYDOWN	Была нажата специальная клавиша
WM_SYSKEYUP	Была отпущена специальная клавиша

Сообщение от таймера:

WM_TIMER	Истекло заданное время
----------	------------------------

Предупреждение: сообщения от таймера помещаются в очередь так же, как и сообщения от мыши и клавиатуры (т.е. таймеры в системе Windows не являются точными). Это объясняется тем, что в противном случае сообщение от таймера, управляемое прерываниями, будет противоречить концепции Windows как многозадачной системе без вытеснений. Таймер полезен тем, что позволяет узнать, когда истекло минимальное время.

Теперь коснемся вопроса различия между сообщениями от мыши, генерируемыми в "клиентной" области окна и генерируемыми в "неклиентной" области. (Напомним, что "неклиентная" область окна включает его границу, системное меню, заголовок окна, меню и т.д.) Вообще, прикладная программа обращает внимание на сообщения от мыши только в "клиентной" области и позволяет оконной процедуре, устанавливаемой по умолчанию, обрабатывать сообщения от мыши в "неклиентной" области.

Сообщение WM_MOUSEMOVE заслуживает особого внимания. Сообщение WM_MOUSEMOVE обрабатывается особым образом, чтобы избежать переполнения очереди аппаратных событий. Дело в том, что при быстром перемещении мыши могут произойти сотни прерываний от мыши. Чтобы этого не случилось, система Windows хранит только одно сообщение о перемещении мыши в каждый момент времени. Когда приходит новое сообщение, система Windows проверяет, есть ли уже подобное. Если оно есть, в старом сообщении обновляется информация о положении мыши. Новая запись делается только в том случае, если сообщения о перемещении мыши еще не было.

Есть два основных типа сообщений от клавиатуры: обычные и системные. Вообще, Вы можете игнорировать системные сообщения от клавиатуры, которые имеют такие имена, как WM_SYSCHAR и WM_SYSKEYDOWN. Оконная процедура, устанавливаемая по умолчанию, превратит их в соответствующую системную команду. Обычные сообщения от клавиатуры предназначены для использования в прикладной программе. Мы обсудим детали всех сообщений от клавиатуры в главе 15, когда речь пойдет о получении входных данных от пользователя.

Сообщения об организации окна

В этой группе 27 сообщений. Похоже, что здесь собраны самые "хитрые" сообщения, поскольку Вам придется потратить много времени на изучение сути каждого. К сожалению, имена сообщений не окажут Вам большой помощи.

Начнем с того, что есть три типа сообщений об организации окна: уведомление, требование действия и запрос. В табл.5.3 представлен перечень всех сообщений об организации окна, разделенных на три типа.

Таблица 5.3. Сообщения об организации окна.

Оконные сообщения: Уведомление

WM_ACTIVATE	Окно активно
WM_ACTIVATEAPP	Прикладная программа активна
WM_CREATE	Окно было создано
WM_DESTROY	Окно было уничтожено
WM_ENABLE	Был разрешен ввод в окно
WM_KILLFOCUS	Окно потеряло управление с клавиатуры
WM_MOUSEACTIVATE	Уведомляет окно о том, что оно станет активным после нажатия кнопки мыши
WM_MOVE	Окно было перемещено
WM_SETFOCUS	Окно получило управление с клавиатуры
WM_SIZE	Окно изменило размер

Оконные сообщения: Требование действия

<code>WM_CLOSE</code>	Заккрыть (уничтожить) окно
<code>WM_ERASEBKGND</code>	Удалить фон
<code>WM_ICONERASEBKGND</code>	Удалить фон окна, показываемого в виде пиктограммы
<code>WM_NCACTIVATE</code>	Изменить заголовок окна, для того чтобы показать активное состояние
<code>WM_NCCREATE</code>	Создать "неклиентную" область данных
<code>WM_NCDESTROY</code>	Уничтожить "неклиентную" область данных
<code>WM_NCPAINT</code>	Перерисовать "неклиентную" область
<code>WM_PAINT</code>	Перерисовать "клиентную" область
<code>WM_PAINTICON</code>	Перерисовать "клиентную" область окна, показываемого в виде пиктограммы
<code>WM_SETREDRAW</code>	Запретить перерисование окна
<code>WM_SETTEXT</code>	Изменить оконный текст
<code>WM_SHOWWINDOW</code>	Изменить видимость окна

Оконные сообщения: Запрос

<code>WM_GETMINMAXINFO</code>	Каковы минимальные/максимальные размеры окна?
<code>WM_GETTEXT</code>	Какой оконный текст?
<code>WM_GETTEXTLENGTH</code>	Какова длина оконного текста?
<code>WM_NCCALCSIZE</code>	Насколько большой должна быть "клиентная" область?
<code>WM_QUERYNEWPALETTE</code>	Есть ли у вас новая палитра?
<code>WM_QUERYOPEN</code>	Может ли быть открыто окно, показываемое в виде пиктограммы?

Сообщение-уведомление говорит оконной процедуре о том, что состояние окна было изменено. Оконная процедура вовсе не должна выполнять действие, обозначенное именем сообщения. Сообщение `WM_MOVE`, например, не является запросом на какое-либо перемещение. Наоборот, оно является "сообщением после действия" и напоминает, что окно уже было перемещено. Стоит отметить, что из сообщения нам не ясно, как именно окно было перемещено. Может быть, пользователь переместил окно с помощью мыши. Или, возможно, это сделала другая программа. В любом случае сообщения-уведомления представляют собой одностороннюю связь с оконной процедурой.

Сообщение-требование действия говорит само за себя. Отсутствие действия вызывает ошибку в интерфейсе пользователя. Например, сообщение `WM_PAINT` посылается оконной процедуре, если окно было "разбито" (утеряна достоверность информации в окне) и его нужно перерисовать. Если оконная процедура не "починит" окно, оно останется "разбитым". Обычно оконная процедура, устанавливаемая по умолчанию, обеспечивает необходимое минимальное действие. Однако иногда бывает нужно перехватить и обработать одно из этих сообщений самому, как в случае с сообщением `WM_PAINT`, которое мы только что рассматривали. При этом убедитесь, что Вы воспроизвели все действия обработчика, устанавливаемого по умолчанию. Чтобы лучше все это понять, посмотрите е-запрос требует ответа. Это сообщение используется для двусторонней связи между исходный текст `DefWindowProc`.

Сообщения Windows и Вашей программой. Подобно сообщениям - требованиям действия, оконная процедура, устанавливаемая по умолчанию, даст приемлемый от-

вет в большинстве случаев. Вы можете перехватить его, если хотите изменить ответ по умолчанию. Например, Вы получите сообщение WM_QUERYOPEN, когда пользователь пытается открыть минимизированное окно. Если Вы хотите, чтобы программа выполнялась только тогда, когда окно появляется в виде пиктограммы, Вам нужно просто ответить FALSE вместо значения TRUE по умолчанию.

Сообщения об организации интерфейса пользователя

Эта группа содержит сообщения для других объектов пользовательского интерфейса, включая меню прикладной программы, указатель мыши, линейку прокрутки, блоки диалога и элементы управления блоком диалога. Она также включает группу сообщений, используемых для поддержки многодокументного интерфейса (MDI). Посмотрите на табл.5.4. Это соглашение о пользовательском интерфейсе, которое впервые появилось в программе табличных вычислений Excel фирмы Microsoft. Соглашение стало чем-то вроде стандарта, так что система Windows теперь имеет встроенную поддержку MDI. Вопросы использования MDI не входят в предмет рассмотрения этой книги, но мы представим Вам другие типы сообщений: сообщения от меню - в главе 11, сообщения от указателя мыши - в главе 16, сообщения линейки прокрутки - в главе 20 и сообщения от блока диалога - в главе 14.

Таблица 5.4. Сообщения об организации интерфейса пользователя.

Сообщения от меню

WM_COMMAND	Был выбран пункт меню
WM_INITMENU	Инициализировать линейку меню
WM_INITMENUPOPUP	Инициализировать всплывающее меню
WM_MENUCHAR	Мнемоническая клавиша использована для выбора меню
WM_MENUSELECT	Пользователь просматривает меню

Системные команды: системное меню, кнопки Min/Max, заголовок окна и т.п.

WM_SYSCOMMAND	Была выбрана системная команда
---------------	--------------------------------

Сообщения от указателя мыши

WM_NCHITTEST	Запрос: где находится мышь в окне?
WM_SETCURSOR	Требование: привести указатель к требуемой форме

Сообщения линейки прокрутки

WM_HSCROLL	Была нажата вертикальная линейка прокрутки
WM_VSCROLL	Была нажата горизонтальная линейка прокрутки

Сообщения блока диалога и элементов управления блоком диалога

WM_COMMAND	Управляющая связь с блоком диалога
WM_COMPAREITEM	Посылается родителю пол с запросом на сравнение двух элементов для сортировки
WM_CTLCOLOR	Управляющий запрос на установку цветов
WM_DELETEITEM	Уведомление пользовательского блока списка или комбинированного блока о том, что элемент был удален
WM_DRAWITEM	Запрос родителю пользовательского элемента управления или пользовательского меню на рисование
WM_GETDLGCODE	Управление запросом: нужен ли ввод с клавиатуры?

WM_GETFONT	Управление запросом: какой шрифт вы используете?
WM_INITDIALOG	Инициализировать диалог
WM_MEASUREITEM	Запрос родителю пользовательского элемента управления или пользовательского элемента на обеспечение размерностей элемента, который должен быть нарисован
WM_SETFONT	Запрос на управление: использовать этот шрифт

Сообщения многодокументного интерфейса (MDI)

WM_CHILDACTIVE	Уведомляет порождающее окно о том, что порожденное окно активно
WM_MDIACTIVATE	Уведомляет порожденное окно MDI о том, что оно получает или теряет активизацию
WM_MDICASCADE	Запрос на организацию открытых порожденных MDI-окон в форме каскадного, ступенчатого расположения (с перекрытиями)
WM_MDICREATE	Требуется, чтобы "клиентное" окно MDI уничтожило порожденное окно MDI
WM_MDIGETACTIVE	Запрашивает "клиентное" окно MDI об активном в настоящее время порожденном окне MDI
WM_MDIICONARRANGE	Требуется организации порожденных окон MDI, показываемых в виде пиктограммы в упорядоченном виде
WM_MDIMAXIMIZE	Запрос на максимизацию порожденного окна MDI, чтобы оно заняло всю "клиентную" область родителя
WM_MDINEXT	Запрос на активизацию следующего порожденного окна MDI
WM_MDIRESTORE	Запрос на восстановление порожденного окна MDI в виде пиктограммы в его предыдущее нормальное или максимизированное состояние
WM_MDISETMENU	Настраивает меню окна MDI с рамкой
WM_MDTITLE	Запрос на организацию открытых порожденных MDI-окон в форме черепичного расположения (без перекрытий)

Сообщения о завершении

Эта группа сообщений самая малочисленная (табл.5.5). Но эти сообщения очень важны, так как они используются для управления завершением программы для Windows (WM_QUIT) и для управления процессом завершения системы (WM_QUERYENDSESSION и WM_ENDSESSION). Сообщение WM_QUIT мы уже рассматривали ранее.

Таблица 5.5. Сообщения о завершении.

Завершение прикладной программы и системы

WM_QUIT	Требование о том, что программа должна завершиться
WM_QUERYENDSESSION	Запрос: готовность к закрытию системы?
WM_ENDSESSION	Уведомление о результатах запроса на закрытие

Частные сообщения

Частные оконные сообщения предназначены для использования с определенным оконным классом. Предопределенные частные сообщения в WINDOWS.H использу-

ются со следующими оконными классами: редактор, экранная кнопка, блок списка и комбинированный блок.

Метод использования частных сообщений predeterminedными оконными типами. Вы могли бы применить для собственных окон, если бы обнаружили, например, что существующие сообщения не обеспечивают необходимых функциональных возможностей. Очень просто определить Ваши собственные типы частных сообщений и использовать их для связи между различными окнами, которые Вы создаете. И хотя уже были определены 250 типов сообщений, переменные для сообщений являются целыми без знака, это означает, что есть место для 65535 типов различных сообщений. При определении частного сообщения Вам необходимо использовать область значений, начинающуюся с WM_USER, которая определена в WINDOWS.H следующим образом:

```
#define WM_USER 0x0400
```

Если бы мы написали оконную процедуру для окон, которые показывают числа, мы могли бы контролировать появление чисел с помощью следующих частных сообщений (PM_ - это венгерский префикс "частного сообщения" - "private message"):

```
#define PM_DECIMAL      WM_USER + 0
#define PM_BINARY       WM_USER + 1
#define PM_HEX          WM_USER + 2
#define PM_OCTAL        WM_USER + 3
#define PM_NODECIMAL    WM_USER + 4
#define PM_DOLLARS      WM_USER + 5
#define PM_WITHCOMMAS   WM_USER + 6
```

Уведомление о системном ресурсе

Есть восемь сообщений с уведомлением о системном ресурсе (табл.5.6). Они посылаются окну верхнего уровня каждой программы при изменении системного ресурса. Например, когда добавляются или удаляются шрифты в системе, распространяется сообщение WM_FONTCHANGE. Когда пользователь изменяет системные цвета или системное время из программы Control Panel, отправляются сообщения WM_SYSCOLORCHANGE или WM_TIMECHANGE. Типичной реакцией на уведомление является запись изменения.

Таблица 5.6. Сообщения с уведомлением о системном ресурсе.

Сообщения с уведомлением о системном ресурсе

WM_COMPACTING	Уведомление о том, что недостаточно системной памяти и что Менеджер Памяти пытается освободить некоторое количество памяти
WM_DEVMODECHANGE	Изменилась настройка принтера
WM_FONTCHANGE	Изменились установленные шрифты в системе
WM_PALETTECHANGED	Изменилась цветная палитра аппаратного обеспечения
WM_SPOOLERSTATUS	Задание было удалено из очереди буфера печати
WM_SYSCOLORCHANGE	Изменился один или более системных цветов
WM_TIMECHANGE	Изменилось системное время
WM_WININICHANGE	Изменился файл инициализации WIN.INI

Не всякое изменение представляет интерес для каждой программы. Например, программа, использующая таймер, вероятно будет проверять новое значение времени при получении сообщения WM_TIMECHANGE. Но если она не использует различные шрифты, то, вероятно, сообщение WM_FONTCHANGE будет проигнорировано.

Поскольку большинство программ используют системные цвета, почти все программы будут реагировать на сообщение WM_SYSCOLORCHANGE. Это является уведомлением об изменении одного или нескольких системных цветов. Обычно системные цвета изменяются программой Control Panel по указанию пользователя. Когда изменение произошло, Control Panel посылает сообщение WM_SYSCOLORCHANGE. При получении этого сообщения программы, использующие системные цвета, реагируют перерисовыванием новыми цветами.

Сообщения о совместном использовании данных

В системе Windows есть сообщения, которые посылаются при совместном использовании данных для поддержки механизмов совместного использования данных - Clipboard и динамического обмена данными (DDE). Полное обсуждение этого вопроса выходит за рамки книги, тем не менее, в табл.5.7 Вам представлен список сообщений о совместном использовании данных в системе Windows.

Таблица 5.7. Сообщения о совместном использовании данных.

Сообщения от буфера вырезанного изображения

WM_ASKCBFORMATNAME	Запрашивает имя формата Clipboard
WM_CNANGECBCHAIN	Уведомление об изменении цепочки просмотра
WM_DESTROYCLIPBOARD	Содержимое Clipboard удаляется
WM_DRAWCLIPBOARD	Содержимое Clipboard изменено
WM_HSCROLLCLIPBOARD	Горизонтальная прокрутка Clipboard
WM_PAINTCLIPBOARD	Запрашивает рисование Clipboard
WM_RENDERALLFORMATS	Запрашивает данные для всех форматов Clipboard, которые были "обещаны"
WM_RENDERFORMAT	Запрашивает данные для одного формата Clipboard, который был "обещан"
WM_SIZECLIPBOARD	Уведомляет владельца пользовательских данных Clipboard об изменении размера окна Clipboard
WM_VSCROLLCLIPBOARD	Вертикальная прокрутка Clipboard

Сообщения от динамического обмена данными (DDE)

WM_DDE_ACK	Подтверждение приема
WM_DDE_ADVISE	Запрос от пользователя DDE об установлении постоянного канала передачи данных
WM_DDE_DATA	Направляет элемент данных от DDE-сервера пользователю DDE
WM_DDE_EXECUTE	Дает указание DDE-серверу выполнить серию команд
WM_DDE_INITIATE	Вход в DDE-сервер
WM_DDE_POKE	Запрос пользователя серверу об обновлении определенного элемента данных
WM_DDE_REQUEST	Однократный запрос пользователя DDE на информацию
WM_DDE_TERMINATE	Выход из DDE-сервера
WM_DDE_UNADVISE	Завершить постоянный канал передачи данных, инициализированный сообщением WM_DDE_ADVISE

Внутренние системные сообщения

Большая группа сообщений определена в `WINDOWS.H`, но не описана ни в какой документации. Это внутренние системные сообщения, последняя группа, которую мы рассмотрим. Система Windows использует их в своих собственных целях.

Внутренние сообщения передаются оконной процедуре. Если оконная процедура его не обработает, оно передается обработчику сообщений, устанавливаемому по умолчанию.

Поскольку у Вас есть исходный текст обработчика сообщений, устанавливаемого по умолчанию, Вы можете подумать о переделке некоторых из этих сообщений и использовании их для Ваших собственных целей. Но будьте осторожны! В будущих версиях Windows может измениться способ использования внутренних сообщений, некоторые из них могут быть вообще удалены. Исходя из опыта, недокументированные возможности интересно изучать, но опасно использовать их в программном обеспечении, предназначенном для широкого распространения.

Часть 3. Введение в интерфейс графических устройств (GDI)

Глава 6. Обзор интерфейса графических устройств (GDI)

В этой и следующих нескольких главах мы подробно остановимся на вопросах, связанных с созданием графического вывода, обсудим различные типы графического вывода, поддерживаемые интерфейсом графических устройств (Graphics Device Interface - GDI). Поскольку библиотеки классов OWL фирмы Borland не имеют каких-либо средств поддержки функций GDI, в следующих главах мы сосредоточимся исключительно на GDI. Вашему вниманию будет предложено несколько примеров программ, написанных на языке C++ на основе среды OWL, которые продемонстрируют Вам, в каких случаях уместно применять вызовы GDI.

Начнем с рассмотрения ряда базовых концепций и средств GDI. В дальнейшем познакомим Вас с выводом пикселей, линий, закрашенных областей и текста. Таким образом, получив основные сведения о графическом выводе, Вы поймете действие механизма, с помощью которого Windows усиливает базовые средства попиксельного вывода графической информации.

Интерфейс графических устройств GDI представляет собой библиотеку графического вывода Windows. GDI обеспечивает графический вывод на экран дисплея и на устройства для получения твердых копий, например, принтеры и плоттеры. GDI отвечает за создание отображения каждой линии, буквы или любого графического знака, выводимого программой для Windows. Сама Windows использует GDI при подборке элементов, составляющих пользовательский интерфейс, - окон, пиктограмм, меню, блоков диалога и т.д.

Средства GDI

Интерфейс графических устройств GDI позволяет выводить такие типы графических объектов, как линии, закрашенные фигуры, а также тексты разнообразных форм и размеров. Данная и последующие главы представят Вам базовые концепции и приемы программирования, необходимые для использования предоставляемых GDI возможностей.



Рис.6.1. Примеры линий, закрашенных фигур и текста, вычерчиваемых с помощью GDI: 1 - линии; 2 - прямоугольники; 3 - прямоугольники со скругленными углами; 4 - эллипсы; 5 - текст.

Устройства GDI

Интерфейс графических устройств позволяет выполнять графический вывод на многие устройства: экраны дисплея, лазерные и матричные принтеры, плоттеры и т.п. Для того чтобы GDI работал с конкретным устройством, необходим специальный элемент программного обеспечения - драйвер устройства. Драйвер устройства для GDI преобразует запросы графического вывода в конкретные действия для рисования на конкретном устройстве. Например, когда программа под Windows рисует на экране EGA космический корабль "Спейс Шаттл", для правильного отображения пикселей на экране GDI вызывает драйвер устройства EGA. При выводе же этого рисунка на лазерный принтер GDI вызывает драйвер другого устройства - драйвер принтера. Помимо этого, драйвер устройства предоставляет GDI набор флагов, которые сообщают, какими графическими возможностями обладает данное устройство. Существует пять наборов таких флагов: для кривых линий, прямых, многоугольников, битовых образов и текстов, - которые сообщают GDI, когда можно направить устройству непосредственно запрос высокого уровня, а когда такой запрос надо предварительно преобразовать в последовательность запросов низкого уровня.

Устройство, работающее с GDI, должно, как минимум, уметь рисовать отдельные пиксели и непрерывные линии. Если устройство обладает такими минимальными возможностями, то все остальное GDI возьмет на себя (с помощью драйвера устройства). GDI обладает множеством средств программной эмуляции, которые принимают на входе высокоуровневые запросы графического вывода (например, "нарисовать закрашенный многоугольник") и преобразуют их в последовательность операций с отдельными линиями и пикселями. Такая программная эмуляция - это одна из причин того, что GDI называют аппаратно-независимой графической библиотекой. Для устройств, возможности которых выходят за рамки указанного минимума, таких как принтер PostScript, GDI по флагам возможностей устройства определяет, можно ли непосредственно направить драйверу запрос высокого уровня.

Помимо физических устройств - экранов и принтеров - GDI поддерживает логические, или псевдоустройства. Псевдоустройства служат для хранения изображений. В отличие от физических устройств, которые выводят изображения на определенной аппаратной базе, псевдоустройства позволяют "перехватить" образ изображения в оперативной памяти или на диске. GDI поддерживает два типа псевдоустройств: битовые образы и метафайлы.

В системе Windows битовые образы всегда имеют прямоугольную форму. Они хранят изображение в памяти в таком виде, в каком графические образы хранятся дисплейным адаптером, и обеспечивают быстрое получение копии картинка. Битовые образы используются и для хранения образов, которые нужно быстро выводить на экран. Например, сама Windows использует их для хранения пиктограмм, курсоров и также небольших символов, участвующих в формировании системных меню, пиктограмм максимизации/минимизации, частей линейки прокрутки и даже значка, указывающего на текущую действующую функцию ("галочку") внутри меню.

Кроме того, битовые образы используются для хранения изображений, полученных со сканера, например логотипа компании. Для этого логотип на бумаге "считывают" сканером в память. Сканер переводит в цифровой код изображение на бумаге, делая его пригодным для хранения в формате битового образа. Примером такой работы может служить считывание изображения с бумаги при помощи ручного сканера для помещения в документ, создаваемый программой WinWord.

Метафайлы создаются средством записи-воспроизведения GDI. С точки зрения расходуемой памяти метафайлы предпочтительнее, чем битовые образы, однако работа с ними происходит медленнее.

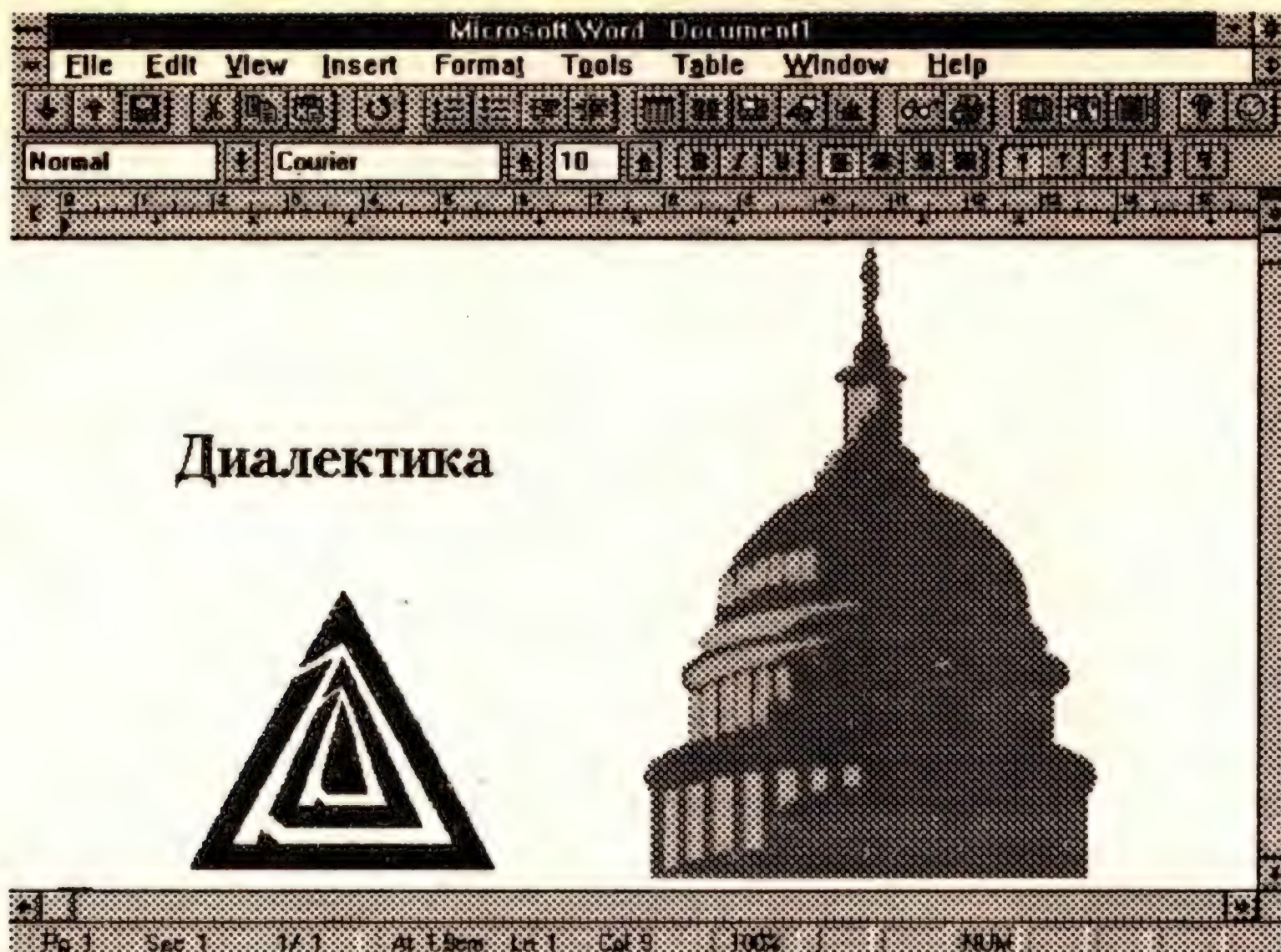


Рис.6.2. Пример сканированного изображения документа, создаваемого с помощью программы WinWord.

Метафайл можно сравнить с магнитофонной лентой. Для создания метафайла Вы "заправляете ленту в магнитофон" и "нажимаете кнопку Запись". Все вызовы GDI будут записываться в метафайл, пока Вы не "нажмете кнопку Стоп". После того как метафайл создан, его можно хранить на диске или передавать в другую программу. Например, часто для передачи метафайла между разными программами используют Clipboard. Для последующего воссоздания рисунка GDI берет "ленту" с метафайлом и воспроизводит его.

Начав программировать с использованием GDI, Вы, вероятно, большую часть усилий потратите на то, чтобы научиться выполнять вывод на физические устройства. Однако имеется ряд тонкостей, которые помогут сделать Вашу работу с экранами и принтерами действительно эффективной. Научившись свободно обращаться с этими устройствами, Вы непременно захотите узнать, как с помощью псевдоустройств улучшить графический вывод Ваших программ.

Программный интерфейс

Из-за большого многообразия поддерживаемых GDI устройств и псевдоустройств эта графическая библиотека могла показаться Вам излишне сложной и трудной в применении. Действительно, некоторые графические библиотеки параллельно ведут два набора подпрограмм: одни для вывода на экран, а вторые - для принтеров. Если бы GDI был построен таким образом, то Вы имели бы четыре разных набора подпрограмм графического вывода: для экранов, принтеров, битовых образов и метафайлов.

Однако GDI построен иначе и поддерживает единый набор подпрограмм для устройств всех типов. Например, подпрограмма `SetPixel` выводит один пиксель на всех поддерживаемых GDI устройствах. Подпрограмма `PolyLine` рисует многоугольник на любом устройстве или псевдоустройстве GDI. В данной главе мы будем вести речь в основном о выводе на экран. Тем не менее, изложенные сведения верны и для устройств других типов, работающих с GDI.

Координаты графического вывода

Прежде чем Вы сможете воспользоваться GDI для создания какого-либо графического вывода, Вам необходимо понять, каким образом он интерпретирует координаты. GDI предоставляет Вам некоторые средства управления этими координатами. Например, Вы можете установить желаемую единицу координатной сетки в дюймах, миллиметрах или каких-либо других смешанных единицах. Для более полного контроля над графическим выводом GDI дает возможность задавать такие единицы, которые наиболее полно соответствовали бы естественным пикселям конкретного устройства. Для обозначения различных поддерживаемых данным интерфейсом систем координат GDI использует термин "режим отображения". Восемь имеющихся режимов отображения GDI показаны в табл.6.1.

Таблица 6.1. Режимы отображения GDI.

Имя режима	Логическая единица	Дюймы	Миллиметры
<i>MM_TEXT</i>	1 пиксель		
<i>MM_HIMETRIC</i>	0.001 мм	0.000394	0.001
<i>MM_TWIPS</i>	1/1440 дюйма	0.000694	0.0176
<i>MM_HIENGLISH</i>	0.001 дюйма	0.001	0.0254
<i>MM_LOMETRIC</i>	0.01 мм	0.00394	0.01
<i>MM_LOENGLISH</i>	0.01 дюйма	0.01	0.254
<i>MM_ISOTROPIC</i> <i>MM_ANISOTROPIC</i>	Масштабирование основано на соотношении между двумя значениями атрибутов контекста устройства: размерами окна экрана и окна данных		

В данной главе мы ограничимся режимом отображения по умолчанию *MM_TEXT*. В этом режиме единица связана с пикселем, т.е. самым малым элементом, отображаемым устройством. Такой режим обеспечивает максимальный контроль над графическим выводом и позволяет избежать ошибок округления, возникающих в остальных режимах. Он часто используется программами, требующими абсолютной точности графического вывода. Например, Aldus PageMaker - программа для макетирования книг - использует режим отображения *MM_TEXT*, поскольку он гарантирует, что соотношения объектов на экране будут правильно выдержаны при выводе на принтер.

Однако у *MM_TEXT* есть и недостаток - он требует некоторых дополнительных усилий, чтобы избежать привязанности программы к конкретному устройству. Остальные режимы отображения, напротив, обладают тем преимуществом, что позволяют писать программы, абсолютно не зависящие от аппаратного обеспечения.

Преимуществом координатной сетки *MM_TEXT* является ее идентичность координатной сетке мыши. Эту систему называют координатной системой "клиентной" области. В координатах "клиентной" области точка начала координат (0,0) находится в верхнем левом углу этой области. В отличие от декартовой системы координат, известной Вам со школьной скамьи, значение Y здесь растет при движении вниз, а X - вправо по сетке.

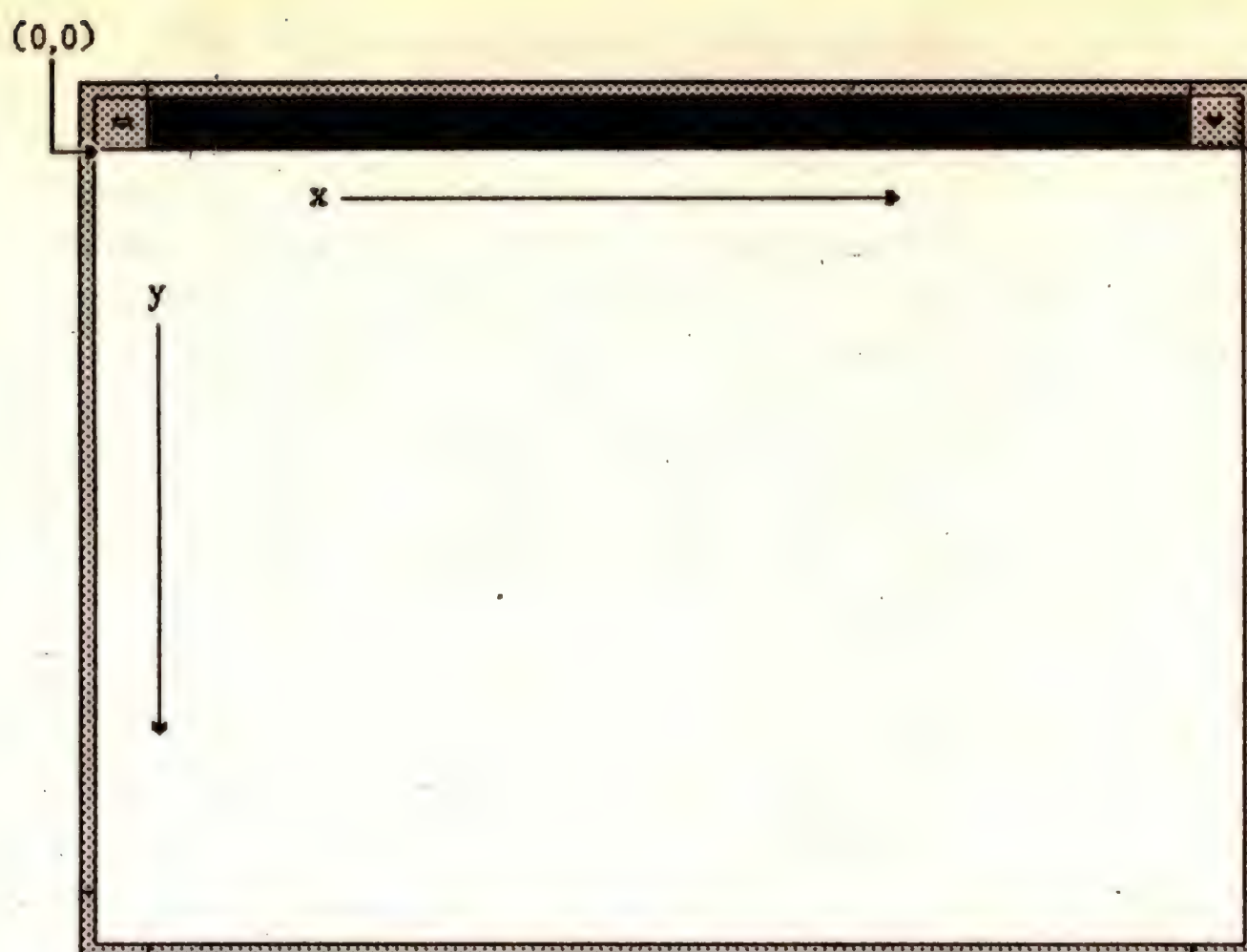


Рис.6.3. Используемая по умолчанию система координат GDI.

Логические объекты графики

Одним из средств достижения аппаратной независимости GDI является использование логических объектов графики. Такой объект описывает, каким образом должен выполняться вывод, это высокоуровневый аппаратно-независимый запрос. GDI поддерживает следующие логические объекты графики: перья (для рисования линий), кисти (для закрашивания областей), шрифты (для вывода текстов) и логические цвета (описывающие цвета вывода).

После создания логического объекта он может быть использован в отношении любого устройства. Однако результаты на каждом из этих устройств могут оказаться сугубо индивидуальными, в зависимости от природы этих устройств. Например, красное перо нарисует красную линию на 16-цветном EGA-дисплее, но черную - на монохромном матричном принтере. Драйвер каждого устройства сам отвечает за интерпретацию логического объекта графики способом, соответствующим реальным возможностям этого устройства.

Представляя Вам каждый тип графического примитива GDI, мы опишем использование различных логических объектов графики и приведем соответствующие примеры программ.

Контекст устройства

Чтобы получить представление о том, как GDI работает с устройствами различных типов, рассмотрим одну из подпрограмм графического вывода GDI - TextOut. Эта подпрограмма выводит на любое поддерживаемое GDI устройство одну строку текста. Выведем слово "Привет!":

```
TextOut(hdc, 10, 10, "Привет!", 5);
```

Обратите внимание на первый параметр `hdc`. Это принятый в Венгерской Записи префикс важного типа данных GDI - интерфейс логического номера контекста устройства. Вспомните, что мы говорили выше о логических номерах: они представляют собой 16-битовое целое без знака и служат для идентификации различных объектов системы. Идентифицируемый данным логическим номером объект играет ключевую роль при выполнении программой для Windows графического вывода на любое устройство, это контекст устройства.

Контекст устройства является комбинацией нескольких понятий. Это как бы набор чертежных инструментов, привязанный к конкретному устройству и позволяющий различным программам использовать его посредством GDI. Интерфейс никогда

не предоставляет программе прямой доступ к контексту устройства. Вместо этого программе передается логический номер, идентифицирующий конкретный контекст устройства. Как и TextOut, каждая подпрограмма графического вывода GDI принимает контекст устройства в качестве первого параметра.

В качестве набора чертежных инструментов контекст устройства представляет собой некоторое множество атрибутов графического вывода, в которое входит одно перо для рисования линий, одна кисть для закрашивания областей и один шрифт для вывода текстов. В любой момент Вы можете изменить контекст устройства, например, установив красные линии, зеленый цвет закрашивания областей и полужирный шрифт для вывода текстов. Вместе взятые, атрибуты графического вывода дают Вам полный контроль над тем, как выглядит и где выполняется графический вывод программы.

Каждый контекст устройства включает в себя 20 атрибутов графического вывода. Храня эти атрибуты в контексте устройства, GDI "скрывает" от Вас всю сложность управления графикой. Такой подход имеет и недостаток, но, согласитесь, он вполне компенсируется упрощением работы с GDI.

А недостаток этот состоит в том, что пока Вы не познакомитесь с атрибутами графического вывода ближе, они могут Вас запутать. Например, рассмотрим подпрограмму TextOut. Введем комментарии к параметрам:

```
TextOut (hdc, /* Логический номер контекста устройства */
10, /* X-координата текста */
10, /* Y-координата текста */
"Привет!", /* Выводимый текст */
5); /* Длина текстовой строки */
```

Можно ли отсюда понять, каким цветом будет выведен текст - красным, синим или черным? Какой будет выбран шрифт, будет ли это полужирный Times Roman из 14 точек или же курсив Helvetica из 24 точек? Поскольку эта информация является частью контекста устройства, ответить на эти вопросы на основании того, как выглядит обращение к подпрограмме, нельзя.

При выполнении графического вывода TextOut берет атрибуты из контекста устройства. Эти атрибуты определяют вид и координаты выводимого текста (включая его цвет и шрифт). Представляя Вам различные подпрограммы графического вывода, мы будем описывать также атрибуты контекста устройства, используемые этими подпрограммами.

Благодаря тому что атрибуты графического вывода скрыты в контексте устройства, вызов подпрограмм GDI может осуществляться с минимальным числом параметров. Практически это означает экономию усилий при набивке программ, в которых есть вызовы GDI. Кроме того, как правило, в промежутках между вызовами эти графические атрибуты не изменяются. Посмотрите, как мог бы выглядеть вызов TextOut, не используя мы понятие контекста устройства:

```
/* Предполагаемый вызов TextOut без контекста устройства */
TextOut (10, /* X-координата текста */
10, /* Y-координата текста */
"Привет!", /* Выводимый текст */
5, /* Длина текстовой строки */
coFore, /* Цвет переднего плана */
coBack, /* Цвет фона */
hClip, /* Область вырезки */
hPalette, /* Цветовая палитра */
hFont, /* Шрифт выводимого текста */
iSpace, /* Межсимвольный интервал */
mmMapMode, /* Режим отображения */
xyViewExt, /* Протяженность окна данных */
xyViewOrg, /* Начало координат окна данных */
```



```

xyWinExt,      /* Протяженность окна экрана */
xyWinOrg);     /* Начало координат окна экрана */

```

Даже не зная значения каждого из показанных атрибутов, можно сказать, что 15 параметров - это слишком, особенно по сравнению с пятью в действительности принимаемыми TextOut параметрами. Без введения понятия контекста устройства, оказывается, даже поздороваться непросто.

Что же входит в контекст устройства? В табл.6.2 приведены атрибуты графического вывода контекста устройства. Для удобства мы указали типы примитивов, использующих каждый атрибут.

Таблица 6.2. Атрибуты графического вывода, входящие в контекст устройства.

Атрибут графического вывода	Значение по умолчанию	Линии					
		⇩ Закрашенные области					
		⇩ Текст					
		⇩ Растр					
		⇩ Комментарии					
⇩							
Цвет фона	Белый	x	x	x		Перо с назначенным стилем, кисть со штриховкой, текст	
Режим фона	OPAQUE	x	x	x		Двухпозиционный переключатель	
Логический номер кисти	Белая кисть		x		x	Закрашиваемые области	
Начало координат кисти	(0,0)		x		x	Кисти штриховки и закраски серым цветом	
Логический номер области прорисовки	Вся поверхность	x	x	x	x		
Логический номер цветовой палитры	Палитра по умолчанию	x	x	x			
Текущая позиция пера	(0,0)	x				Для подпрограммы LineTo	
Режим графического вывода	R2_COPYPEN	x	x			Логика отображения перекрывающихся изображений	
Логический номер шрифта	Системный шрифт				x		
Межсимвольный интервал	0				x		
Режим отображения	MM_TEXT	x	x	x	x	Одна единица равна одному пикселю	
Логический номер пера	Черное перо	x	x				
Режим закрашивания многоугольников	Альтернативный		x			Для подпрограммы Polygon	
Режим растяжения	Черный по белому				x	Для подпрограммы StretchBlt	
Выравнивание границ текста	По левому и верхнему краям			x			

Цвет текста	Черный для текста и кистей с монохромным шаблоном закрашивания	x	x		Цвет переднего плана
Выравнивание строк текста	0,0		x		Отбрасывание лишних символов
Протяженность окна данных экрана	(1,1)	x	x	x	Отображение координат окна данных и окна экрана
Начало координат окна данных экрана	(0,0)	x	x	x	То же
Протяженность окна экрана	(1,1)	x	x	x	— " —
Начало координат окна экрана	(0,0)	x	x	x	— " —

Не каждый атрибут графического вывода используется его подпрограммами. Например, атрибут цвета текста не играет никакой роли при рисовании линий. Каждая подпрограмма берет из контекста устройства только те атрибуты, которые ей нужны. Рассмотрим и другие аспекты контекста устройства.

Контекст устройства связывает программу с конкретной поверхностью рисования. Например, программа, которой требуется системный экран, должна обратиться к его контексту устройства (как это делается, мы опишем ниже). Для вывода на принтер или псевдоустройство, например, в битовый образ, программа должна получить соответствующий контекст устройства, чтобы установить связь с нужной поверхностью рисования. Такое соединение является логическим, а не физическим.

Чтобы понять, что подразумевается под физическим соединением, рассмотрим, как работают программы для MS-DOS типа Quattro Pro. Когда такая программа рисует на экране, она выполняет запись прямо в буфер видеопамати на плате видеоадаптера. При этом она напрямую манипулирует аппаратными регистрами, чтобы получить нужный рисунок электронной таблицы или график. Для поддержки этих возможностей ей необходим собственный драйвер устройства, чтобы иметь возможность различать карты CGA, VGA и 8514/a. Программа Lotus 1-2-3 позволяет это, так как MS DOS является строго однозадачной системой. Никакой опасности нарушения работы каких-либо других программ не возникает.

Однако Windows - это многозадачная система, и поэтому ни одна программа не может напрямую обратиться к физическому устройству вывода, не нарушив при этом работу других программ. Любая программа должна использовать логические связи, устанавливаемые с устройствами посредством контекста устройства. Этот метод применяется всеми программами для Windows, так что GDI разрешает все конфликты между конкурирующими за ресурс программами, не допуская развала системы. В этом заключается третья исполняемая контекстом устройства роль.

Чтобы избежать конфликтов, связанных с совместным использованием устройств, программа должна получить у контекста устройства "пропуск" к устройству. Работа системы пропусков зависит от типа устройства. На устройствах получения твердых копий это делается путем буферизации вывода, а на видеоустройствах - путем выделения так называемой области прорисовки.

Для принтеров и плоттеров GDI заимствует подход, принятый на больших машинах, - вывод сначала буферизуется на диске. Механизм буферизации вывода Windows, реализуемый менеджером печати, играет роль регулятора, управляющего потоками вывода на устройства получаемых твердых копий. В противном случае вывод одной программы мог бы перемешаться с выводом другой. Чтобы осуществ-

влять буферизированный вывод, контекст устройства помогает GDI содержать порознь отдельные задания для печати.

Для вывода на экран Windows использует другой подход. В понятие области прорисовки изображения входит создание области, выделенной данной программе, для вывода. За границей этой области, "забором", программа рисовать не может.

Представьте себе: у Вашего соседа двор огорожен забором, чтобы его собака не могла никуда убежать. Во дворе эта собака может делать, что пожелает, но забежать на Вашу территорию она не может. GDI делает примерно то же самое. Когда программа хочет нарисовать что-то в окне, она получает у менеджера окон контекст устройства, в который встроен "забор" вокруг "клиентной" области окна. Внутри него программа может рисовать что угодно, однако выйти за его пределы она не может.

Приведем пример того, как работают программы, использующие область прорисовки (рис.6.4). Представьте себе две программы, совместно использующие экран: Notepad и Clock. Прежде чем они начнут работать, вокруг "клиентной" области каждой из них создается "забор", предотвращающий вывод программы вне ее "клиентной" области.

"Забор" Notepad представляет собой прямоугольник. Этот "забор" не дает пользователю забраться на территорию Clock и предотвращает вывод в "неклиентную" (служебную) область самого Notepad. Кроме того, подобно большинству программ для Windows, Notepad сам не занимается организацией своей "неклиентной" области, оставляя эту работу Windows.

Область прорисовки для Clock представляет собой нечто более сложное (так как Notepad расположен поверх Clock). Однако мы знаем, что и здесь секундная стрелка Clock не портит содержимое Notepad. Стрелки Clock ходят под Notepad (рис.6.5). Форма "забора" Clock получается несколько сложнее, нежели прямоугольная форма "забора" Notepad.

Не подумайте только, что Clock каким-либо специальным образом заботится, чтобы не "залезть" в Notepad. На самом деле Clock вообще ничего не знает о существовании Notepad. Все дело здесь в том контексте устройства, который Clock получает от менеджера окон. Когда Clock вызывает прочие графические подпрограммы интерфейса GDI, логический номер контекста устройства позволяет GDI контролировать вывод графической информации, чтобы не выйти за границы "забора", так что со стороны Clock практически никаких действий и не требуется - все делает GDI.

Как GDI производит контроль границ вывода? Рассмотрим более простой случай: каким образом GDI выполняет вырезку по прямоугольной границе клиентной области Notepad. GDI с помощью драйвера устройства устанавливает вокруг нее прямоугольный "забор", определяемый четырьмя координатами углов. Границы его являются частью атрибута графического вывода в контексте устройства, это области вырезки. Каждая подпрограмма графического вывода GDI при рисовании проверяет эти границы. Это означает, что вырезка может произойти прямо посередине буквы. Однако для безупречной работы окон это условие обязательно.

Более сложный случай вырезки мы наблюдаем на примере Clock, находящегося под Notepad. В этом случае GDI определяет вырезку, разбивая окно Clock на четыре прямоугольника, возникающие вокруг области, занимаемой Notepad. Здесь вместо четырех границ одного прямоугольника GDI работает с 16 границами четырех прямоугольных областей. Для этого GDI разбивает каждую операцию графического вывода на четыре отдельные операции, по одной на каждый прямоугольник вырезки.

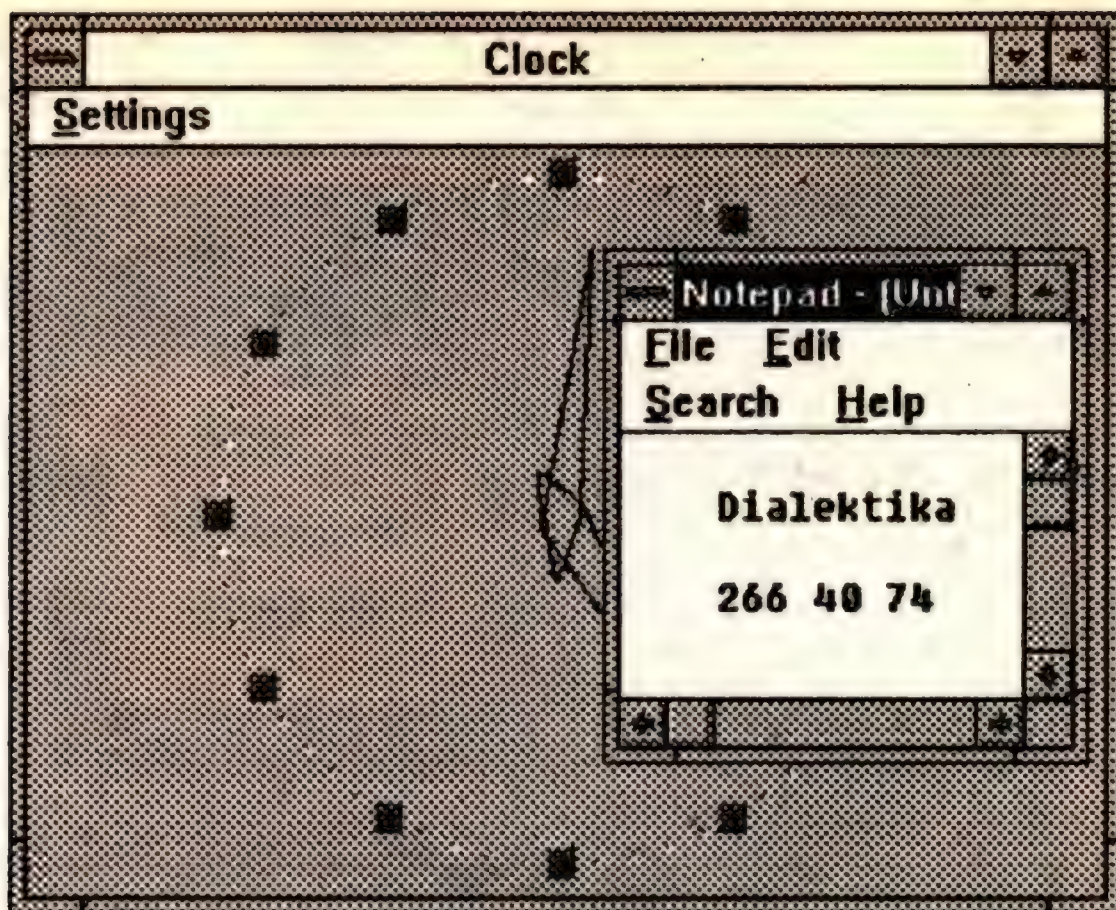


Рис.6.4. Совместное независимое использование экрана программами Notepad и Clock благодаря методу вырезанного изображения.

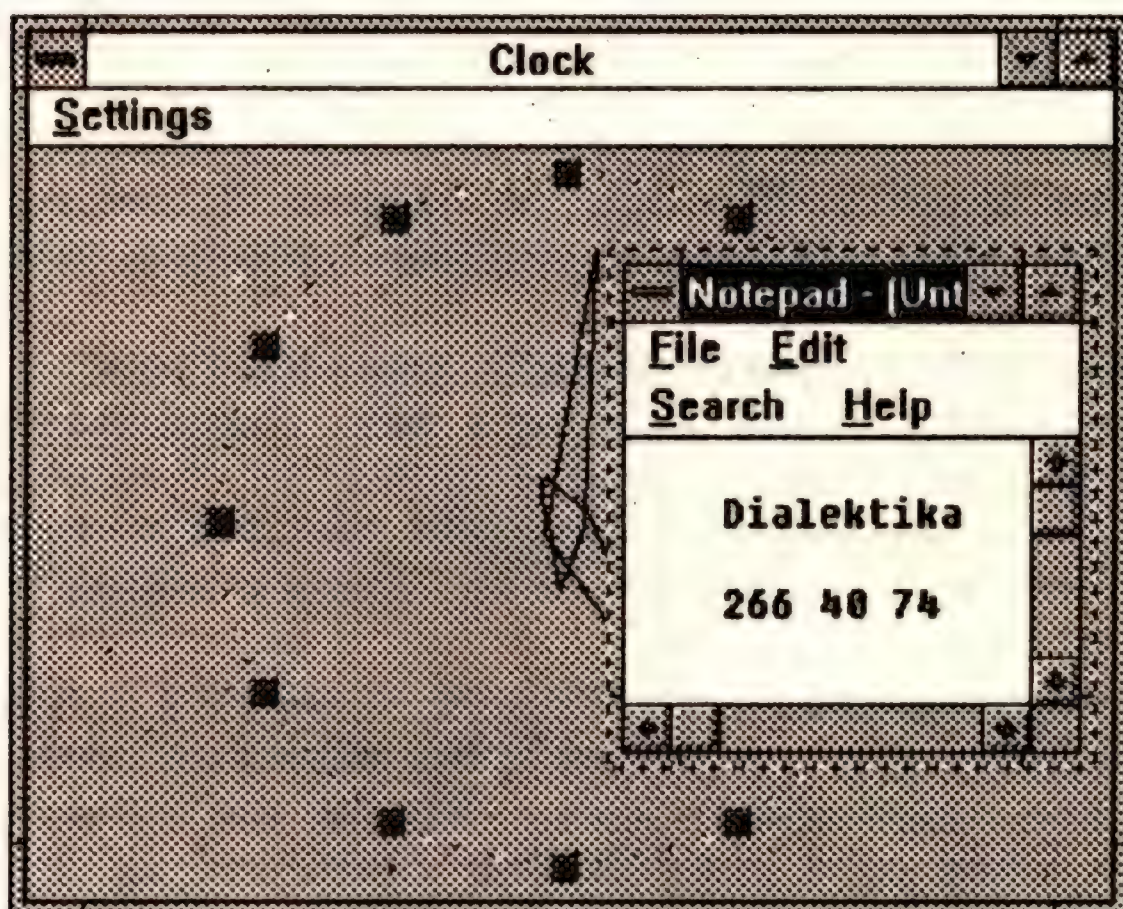


Рис.6.5. Форма забора вокруг окна программы Clock.

Такой сложный алгоритм вырезки дает нам ключ к пониманию того, как GDI хранит информацию об областях вырезки. Набор из одного или более прямоугольников объединяется в структуру данных, которая называется областью вырезки. Этот атрибут является частью контекста устройства, так что вырезку может реализовать любое устройство, работающее с GDI. Область вырезки может включать в себя всю поверхность рисования либо состоять из одного или нескольких прямоугольников.

Хотя функция вырезки относится ко всем устройствам, вырезка на экране представляет собой особый случай. Отсечка, вообще, была изобретена, чтобы помешать одной программе портить области вывода других. Та часть Windows, которая отвечает за вырезку на экране и за создание окон, называется менеджером окон.

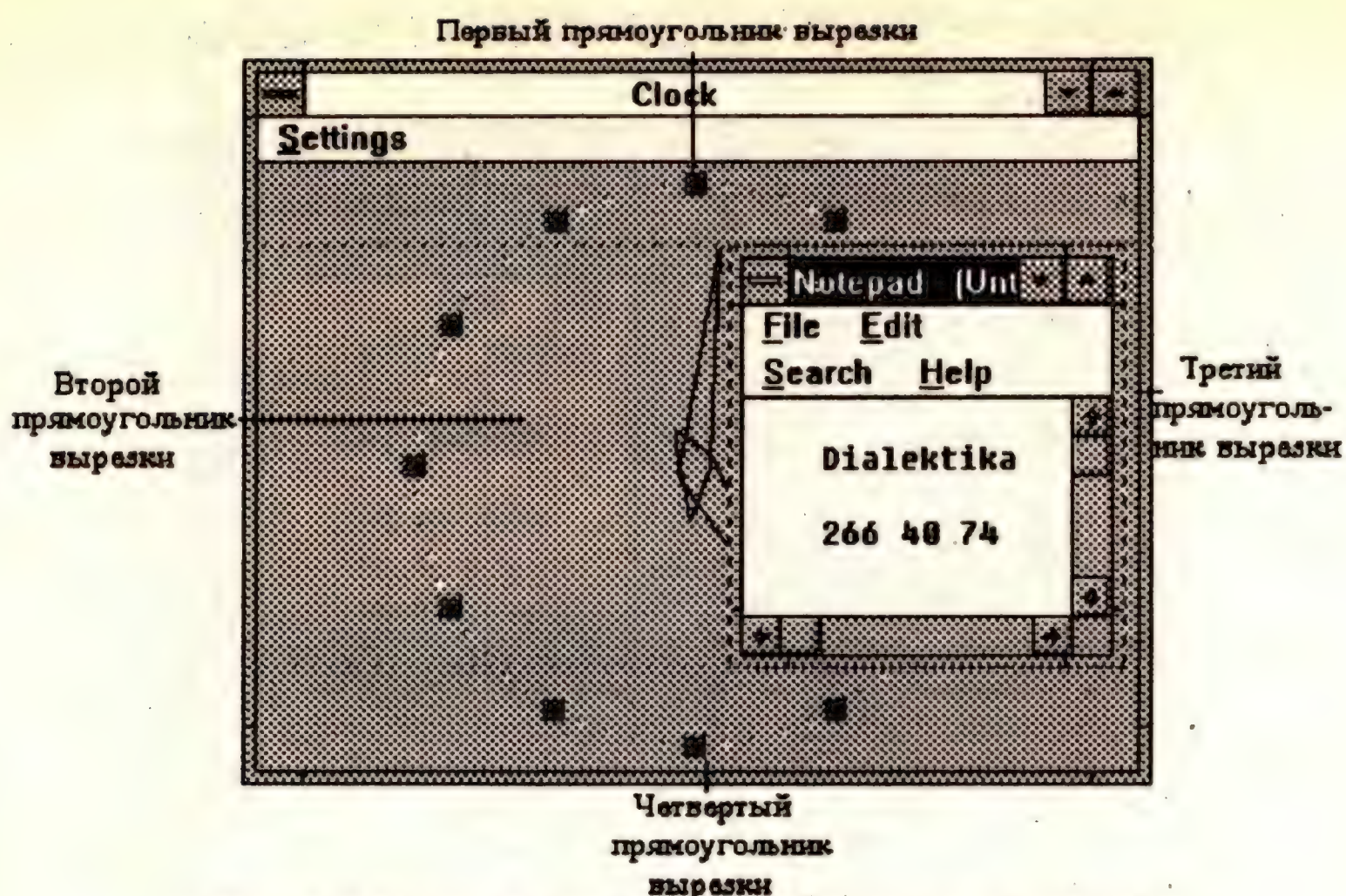


Рис.6.6. Четыре прямоугольника вырезки Clock.

Вырезка изображения и менеджер окон

На примере описанного выше взаимодействия между Notepad и Clock ясно, что именно реализованное как часть GDI средство вырезки сделало возможным создание многооконной системы. Оно позволяет программам разделять экран, не мешая друг другу.

Хотя обслуживают вырезку именно подпрограммы GDI, сам по себе GDI не устанавливает границ окон. Это работа менеджера окон. Представляя менеджер окон в главе 1, мы говорили, что именно он отвечает за пользовательский интерфейс системы. Поскольку весь пользовательский интерфейс связан с распределением окон на экране дисплея, то менеджеру окон приходится "строить заборы" вокруг каждого из этих окон.

Менеджеру окон принадлежит все множество контекстов устройства для рисования на системном дисплее. Если программа хочет нарисовать окно, она заимствует у менеджера один из этих контекстов устройств. Прежде чем передать контекст устройства программе, менеджер окон устанавливает соответствующую область вырезки. Программа может заимствовать у менеджера контекст устройства с помощью трех различных подпрограмм. Для каждой из них в контексте устройства устанавливается своя область вырезки. Эти подпрограммы представлены в табл.6.3 вместе с подпрограммами, которые обеспечивают возврат контекста устройства менеджеру окон. Если программа заимствует у менеджера контекст устройства, она же должна позаботиться о том, чтобы потом вернуть его менеджеру. Ниже Вы увидите, что эту роль играет конструкция программы под названием "сэндвич".

Таблица 6.3. Подпрограммы, заимствующие и возвращающие контекст устройства системного дисплея.

Заимствующая подпрограмма	Возвращающая подпрограмма	Описание
BeginPaint	EndPaint	Вырезка по границе поврежденной (или требующей перерисовки) части клиентной области
GetDC	ReleaseDC	Вырезка по границе всей клиентной области
GetWindowDC	ReleaseDC	Вырезка по границе всего окна (клиентной и неклиентной областей)

Первая из этих подпрограмм, `BeginPaint`, позволяет программе реагировать на запросы, которые менеджер окон делает для рисования в окне или для восстановления поврежденной области окна. Менеджер окон сообщает программе о необходимости восстановить такую область, направляя сообщение `WM_PAINT`. Вы увидите, что данное сообщение играет ключевую роль в организации "клиентной" области окна.

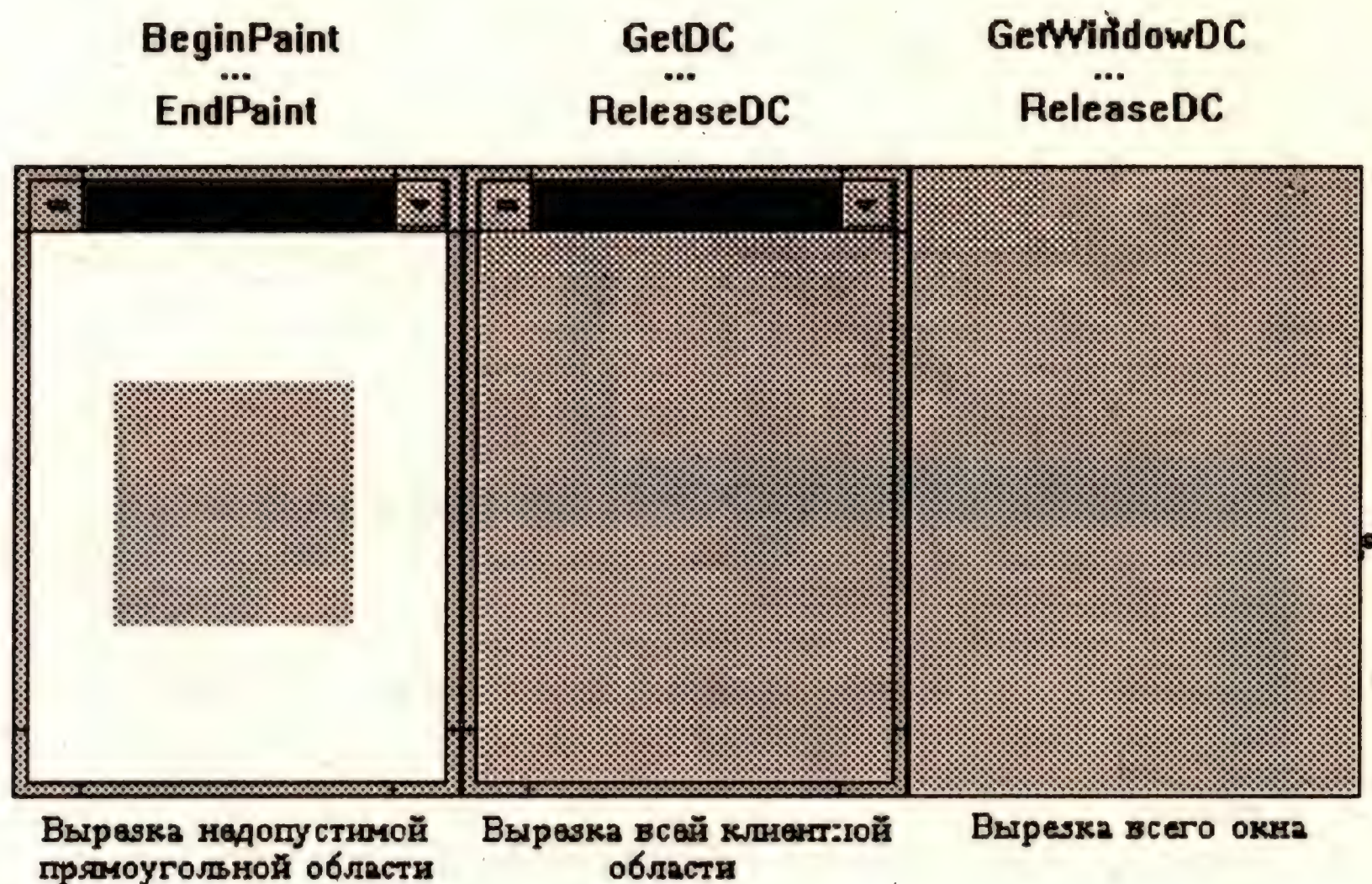


Рис.6.7. Три примера выполнения определения области прорисовки менеджером окон.

Вторая подпрограмма, `GetDC`, дает возможность программе рисовать в "клиентной" области окна. "Забор", по которому выполняется вырезка, позволяет удерживать вывод в пределах "клиентной" области, даже если сверху нее лежит окно другой программы, как было показано на примере пары программ `Clock` и `Notepad`. Эта подпрограмма используется для большинства операций вывода, не связанных с сообщением `WM_PAINT`. Например, в ответ на сообщение `WM_CHAR` нам может понадобиться отобразить на экране введенный символ.

Третья подпрограмма, `GetWindowDC`, позволяет устанавливать границу вырезки таким образом, чтобы можно было рисовать где угодно, включая и "неклиентную" область. Эту программу использует сам Менеджер окон для создания "неклиентной" части окна. Обычно программы под Windows сами не занимаются этой работой, оставляя ее оконной процедуре.

Возможно, прочитав данное введение в GDI, Вы ощутили необходимость ознакомиться с примерами программ, демонстрирующих представленные выше концепции. В следующей главе мы покажем Вам программу, которая всего-навсего выполняет вывод отдельного пикселя, но зато знакомит Вас с одним из наиболее важных сообщений, с которым приходится работать программам для Windows, - `WM_PAINT`. Мы также рассмотрим, как в Windows происходит работа с цветами, и, кроме того, расширим набор примитивов GDI, создав собственный примитив для отображения маркеров.

Глава 7. Пиксели и маркеры

Первая рассматриваемая здесь программа просто отображает на экране один пиксель. Однако при этом она знакомит Вас с важнейшим сообщением, с которым имеет дело любая программа, выполняющая вывод на дисплей - WM_PAINT. Если Вы запустите три экземпляра программы с разными размерами окон, то всякий раз пиксель будет появляться точно в центре каждого из окон (рис.7.1).

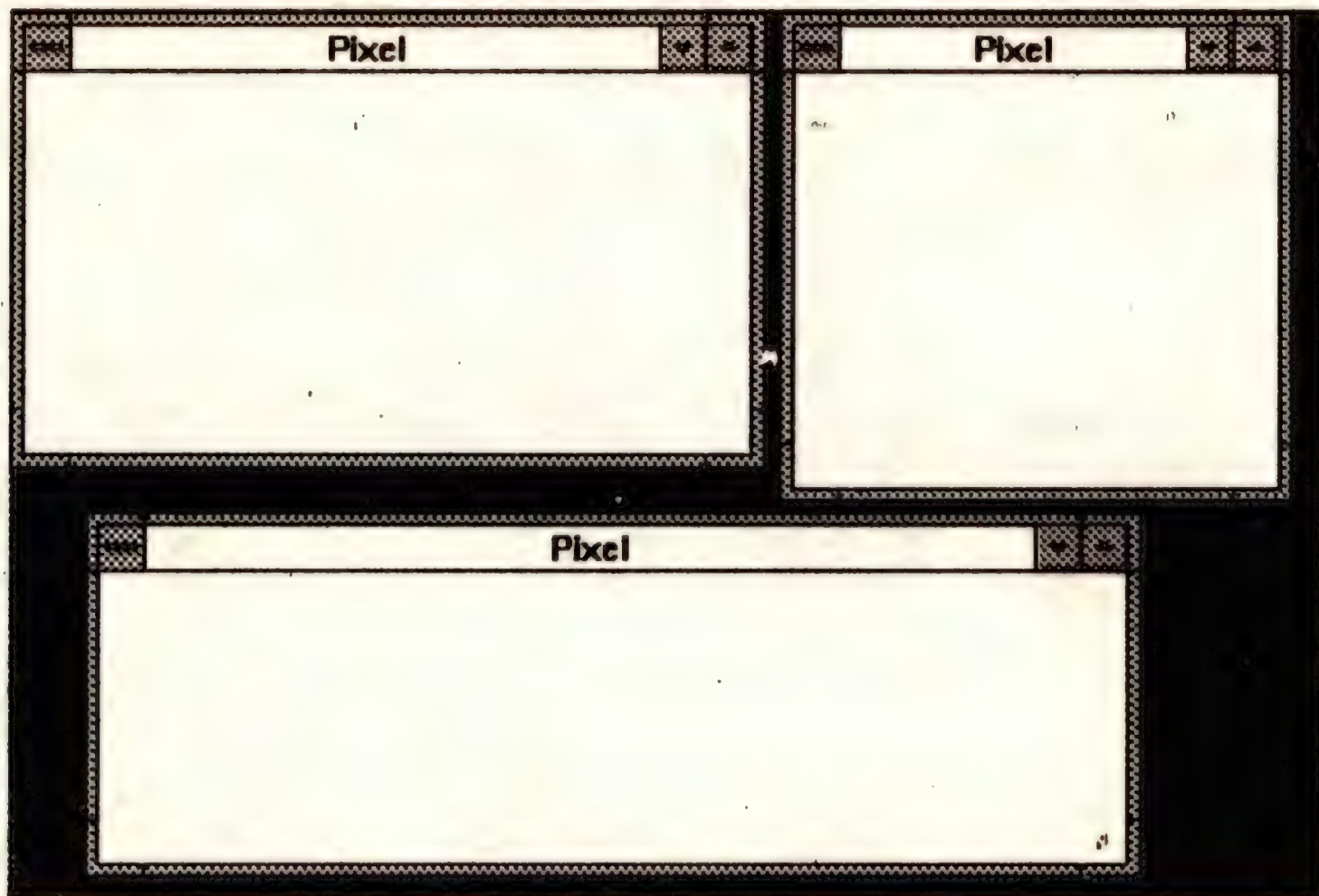


Рис.7.1. Три экземпляра программы Pixel.

Как и для других приводимых в этой книге программ, отправной точкой для программы Pixel служит программа MIN. Ниже приводятся листинги исходных файлов программы.

MAKEFILE.MAK

.AUTODEPEND

Определения транслятора

INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;C:\BORLANDC\INCLUDE

CC = bcc -c -D_CLASSDLL -H -ml -WS -w -I\$(INC)

Неявные правила

.c.obj:

\$(CC) \$(<)

.cpp.obj:

\$(CC) \$(<)

Явные правила

Pixel.exe: Pixel.res Pixel.def Pixel.obj

tlink /c/C/n/P-/Twe/x @Pixel.LNK

RC Pixel.res Pixel.exe

Зависимости отдельных файлов

Pixel.obj: Pixel.cpp

Pixel.res: Pixel.rc Pixel.cur Pixel.ico

RC -R -FO Pixel.res Pixel.RC

PIXEL.LNK

```
c:\borlandc\lib\c0wl.obj+
Pixel.obj
Pixel,Pixel
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwl.lib+
import.lib+
mathl.lib+
cl.lib
Pixel.def
```

PIXEL.CPP

```
/*
PIXEL.CPP  Высвечивается группа пикселей в центре "клиентной" области
*/

#include <owl.h>

/*
Определения классов
*/
class TPixelApplication : public TApplication
{
public:
    TPixelApplication (LPSTR lpszName, HANDLE hInstance,
                       HANDLE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow);
    virtual void InitMainWindow ();
};
class TPixelWindow : public TWindow
{
public:
    TPixelWindow (PTWindowsObject pwParent, LPSTR lpszTitle,
                  PTModule pmModule);
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WNDCLASS&);
    virtual void WMPaint (TMessage& Msg) = [WM_FIRST + WM_PAINT];
};

/*
Основная функция: WinMain
*/
int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
{
    TPixelApplication Pixel ("Pixel", hInstance, hPrevInstance,
                              lpszCmdLine, nCmdShow);
    Pixel.Run();  return Pixel.Status;
}

/*
Компонентная функция класса прикладной программы
*/
TPixelApplication::TPixelApplication (LPSTR lpszName,
                                       HANDLE hInstance, HANDLE hPrevInstance,
                                       LPSTR lpszCmdLine, int nCmdShow)
    :TApplication (lpszName, hInstance,
```



```

        hPrevInstance, lpszCmdLine, nCmdShow)
    {
        /* Здесь инициализируется программа. */
    }

/*
Компонентная функция класса прикладной программы
*/
void TPixelApplication::InitMainWindow ()
{
    MainWindow = new TPixelWindow (NULL, "Pixel", NULL);
}

/*
Компонентная функция класса TPixelWindow
*/
TPixelWindow::TPixelWindow (PTWindowsObject pwParent,
    LPSTR lpszTitle, PTModule pmModule)
    :TWindow (pwParent, lpszTitle, pmModule)
{
    /* Здесь выполняется инициализация окна. */
}

/*
Компонентная функция класса TPixelWindow
*/
LPSTR TPixelWindow::GetClassName ()
{
    return "Pixel:MAIN";
}

/*
Компонентная функция класса TPixelWindow
*/
void TPixelWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
    wc.style=CS_HREDRAW | CS_VREDRAW;
}

/*
Компонентная функция класса TPixelWindow
*/
void TPixelWindow::WMPaint (TMessage&)
{
    int          x,y;
    PAINTSTRUCT  ps;
    RECT         rClient;

    BeginPaint (HWindow, &ps);
    GetClientRect (HWindow, &rClient);
    x = rClient.right / 2;
    y = rClient.bottom / 2;

    SetPixel (ps.hdc, x, y, RGB (0, 0, 0));
    EndPaint(HWindow, &ps);
}

```


PIXEL.RC

```
snapshot icon Pixel.ico  
hand cursor Pixel.cur
```

PIXEL.DEF

```
NAME PIXEL  
EXETYPE WINDOWS  
DESCRIPTION 'Pixel -- SetPixel sample program'  
CODE MOVEABLE DISCARDABLE  
DATA MOVEABLE MULTIPLE  
HEAPSIZE 512  
STACKSIZE 5120
```

Функция WinMain программы PIXEL в целом такая же, как и в MIN. Практически одинаковыми являются коды инициализации ее объекта прикладной программы и оконного объекта. Стоит отметить одно изменение. В определение оконного класса добавлены два новых бита стиля:

```
wc.style=CS_HREDRAW | CS_VREDRAW;
```

Они заставляют Менеджер Окон генерировать сообщение WM_PAINT при изменении размера окна как в вертикальном (CS_VREDRAW), так и в горизонтальном (CS_HREDRAW) направлении. Биты стиля вызывают перерисовку всего окна (посредством сообщения WM_PAINT) при каждом изменении размеров окна, чтобы вывешиваемый пиксель гарантированно находился в центре окна.

Сообщение WM_PAINT является жизненно важным для любой выполняющей вывод на экран программы. При хаотическом и неосторожном обращении с окнами пользователь, подобно слону в посудной лавке, может образовать "дыры" в пользовательском интерфейсе, которые Windows не может залатать самостоятельно. В частности, в случае повреждения "клиентной области" окна возможности Windows по ее восстановлению весьма ограничены.

Существуют ситуации, в которых Windows может предвидеть, что объект (например, блок диалога или меню) на короткое время закроет окно. В этих случаях Windows фотографирует ту часть, которая будет закрыта, чтобы в нужный момент восстановить ее без посторонней помощи.

Однако такие "снимки" требуют большого расхода памяти, так что Windows прибегает к этому средству, только когда данная операция критична с точки зрения быстродействия пользовательского интерфейса. Например, "снимки" закрываемых областей применяются при работе со всплывающими меню. Если бы столь критичная к скорости часть интерфейса работала медленно, это лишило бы смысла всю систему в целом.

В большинстве случаев Windows вызывает Вашу программу для "латания дыр" в пользовательском интерфейсе посредством передачи ей сообщения WM_PAINT. В ответ на WM_PAINT Ваш оконный объект принимает контекст устройства, который, в свою очередь, дает ему "чертежные инструменты", доступ к устройству и "пропуск", необходимые для рисования в окне или восстановления его содержимого.

Подпрограмма BeginPaint

Подпрограмма BeginPaint обеспечивает требуемый контекст устройства. BeginPaint заимствует контекст устройства из источника, поддерживаемого Менеджером Окон. Эта подпрограмма специально ориентирована на сообщение WM_PAINT, так как она устанавливает в контексте устройства область вырезки, соответствующую поврежденной части Вашего окна. Эту область иногда называют недействительной

областью или недействительным прямоугольником (если эта область определена как простой прямоугольник).

Недействительная область может совпадать с "клиентной" областью окна, но обычно занимает только ее часть. Например, при одновременном выполнении программ Clock и Pixel один угол Clock может закрывать часть "клиентной" области Pixel. При нажатии кнопки мыши над Pixel для активизации ее окна закрытая ранее часть "клиентной" области Pixel должна быть перерисована. За это отвечает сообщение WM_PAINT. В возвращаемом BeginPaint контексте устройства устанавливается соответствующая область вырезки. Сделав недействительную область подмножеством общей "клиентной области", Менеджер Окон облегчает "ремонт" поврежденной части окна.

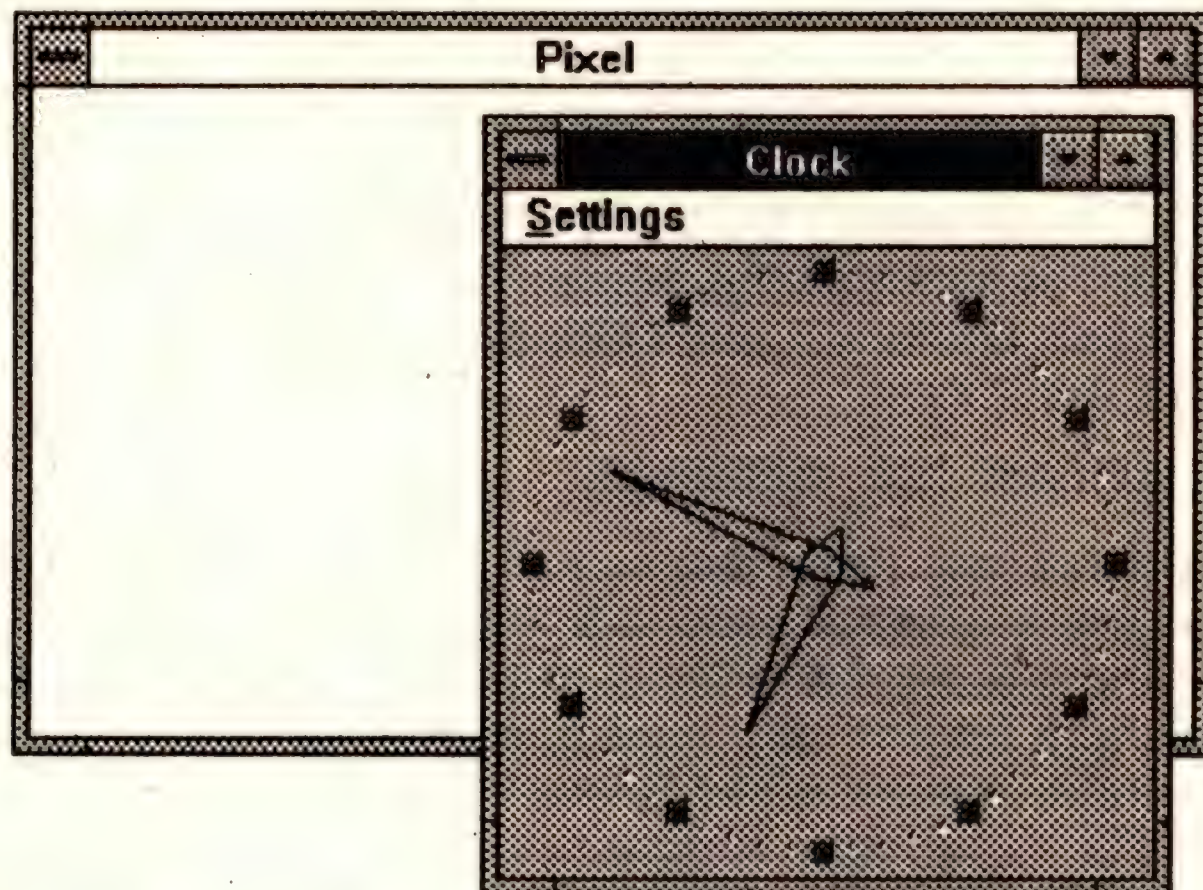


Рис.7.2. Окно программы Clock частично перекрывает окно программы Pixel.

Из предыдущего обсуждения вырезки изображения Вы можете вспомнить, что прямоугольник вырезки позволяет рисовать только в пределах своих границ. Вне "забора" никакое рисование невозможно. Почему функция вырезки устроена именно так?

Вспомните, что сообщение WM_PAINT предназначено для восстановления поврежденных частей окна. Границы вырезки окружают именно ту часть окна, которая нуждается в перерисовке. Это единственная область, в которой программе в данный момент разрешено выполнять вывод. И, с другой стороны, это единственная часть окна, в которой необходимо что-то нарисовать. По определению перерисовывать в ответ на WM_PAINT другие, неповрежденные, части окна нет необходимости.

Рассмотрим поближе обработку в программе сообщения WM_PAINT, что поможет нам лучше разобраться в том, что же при этом происходит.

Подпрограмма BeginPaint принимает два параметра: логический номер окна и указатель на структуру данных PAINTSTRUCT. Логический номер окна идентифицирует окно, в котором выполняется ремонт "клиентной" области.

Структура PAINTSTRUCT определена в WINDOWS.H следующим образом:

```
typedef struct tagPAINTSTRUCT
{
    HDC      hdc;
    BOOL     fErase;
    RECT     rcPaint;
    BOOL     fRestore;
    BOOL     fIncUpdate;
    BYTE     rgbReserved[16];
} PAINTSTRUCT;
```


Из шести полей этой структуры данных полезными являются лишь первое и третье. Первое поле `hdc` - это логический номер контекста устройства, передаваемый подпрограммам графического вывода GDI. Как сказано в "Справочном руководстве по ObjectWindows для C++", второе поле `fErase` - это флаг, задающий необходимость стирания окна. Фактически это поле всегда равно нулю. Четвертое, пятое и шестое поля предназначены для внутреннего использования Windows.

Поле `rcPaint` описывает прямоугольник, ограничивающий поврежденную, или недействительную область нашего окна. Хотя большинство программ игнорирует это поле, некоторые программы могут использовать его, чтобы взять на себя часть выполняемой GDI работы. Например, если программа интенсивно рисовала в окне, она может с помощью этого поля в точности отметить, какая часть окна нуждается в обновлении. Тем самым, заранее ограничив вырезку этим прямоугольником, программа может повысить свое быстродействие.

Подпрограмма `BeginPaint` дает нам логический номер контекста устройства, необходимый для рисования пикселя. Рассмотрим другие подпрограммы, которые наша оконная процедура вызывает в ответ на сообщение `WM_PAINT`.

Подпрограмма `GetClientRect`

Выше мы сравнивали окно с холстом, на котором рисует художник. Однако, в отличие от холста, окно Windows может изменять свой размер. Чтобы предоставить Вашей программе возможность следить за этим изменением, подпрограмма `GetClientRect` позволяет ей определять размер "клиентной" области окна (получение этой информации с помощью сообщения `WM_SIZE` тоже непростое дело). Подпрограмма `GetClientRect` принимает два параметра:

- * Логический номер окна. Мы используем для этого компонент данных `HWindow` с логическим номером окна, наследуемым нашим оконным классом.

- * Указатель на структуру данных о прямоугольнике (`RECT`). В этой структуре возвращается размер "клиентной" области в пикселях.

Структура `RECT` определена в `WINDOWS.H` следующим образом:

```
typedef struct tagRECT
{
    int left;
    int top;
    int right;
    int bottom;
} RECT;
```

Одним из свойств `GetClientRect` является то, что она всегда возвращает нули в полях `left` и `top`, так что их можно смело игнорировать. Для определения размеров "клиентной" области мы используем значения `right` и `bottom`. Для помещения пикселя в центр "клиентной области" мы вычисляем координаты `x` и `y` как `r.right/2` и `r.bottom/2`.

Подпрограмма `SetPixel`

Подпрограмма `SetPixel` принимает четыре параметра:

`SetPixel (hDC, X, Y, crColor);`

- * Логический номер контекста устройства. Мы используем здесь логический номер из `BeginPaint`.

- * Координаты `X` и `Y`. Помещаем пиксель в центр "клиентной" области, вычисляя координаты делением на два значений, полученных из `GetClientRect`.

- * Цвет. GDI обеспечивает три способа задания цвета: в виде трех RGB-составляющих, по индексу в палитре, а также в виде RGB-составляющих, но относительно палитры.

Тройка RGB-составляющих представляет собой три значения базовых цветовых компонентов - красного, зеленого и синего. Эти компоненты задаются как длинное целое без знака, т.е. 4-байтовым, или 32-битовым значением. Три байта из четырех хранят значения интенсивности соответствующих составляющих. Один байт на составляющую означает 256 значений интенсивности для каждой, или около 16 миллионов уникальных комбинаций цвета.

Макропроцедура RGB из WINDOWS.H дает простейший способ создания троек RGB-составляющих. Синтаксис этой макропроцедуры следующий:

```
rgbColor = RGB (bRed, bGreen, bBlue),
```

где bRed, bGreen и bBlue - это целые от 0 до 256.

Макропроцедура RGB упаковывает их в одно длинное целое без знака.

Несмотря на то что общее число различных RGB-комбинаций составляет примерно 16 миллионов, получаемый в действительности цвет зависит от устройства. Например, 16-цветный адаптер EGA отображает RGB-значение ближайшим доступным физическим цветом. Говоря о перьях, мы увидим, как GDI "смешивает краски", эмулируя гораздо большее число цветов, чем их имеется у устройства. Однако пикселю можно задать только чистый, или физический цвет. Гарантированный результат дают лишь две RGB-тройки: RGB(0,0,0) всегда соответствует черному, а RGB(255,255,255) - белому цвету.

Другой способ состоит в задании индекса цвета в палитре. Палитра - это таблица RGB-троек. Поэтому, на первый взгляд, индекс палитры является всего лишь альтернативой значений RGB-тройки.

Тем не менее, в отличие от RGB-троек, индекс палитры позволяет программе задавать точный физический цвет. Например, дисплейные адаптеры VGA и 8514/а аппаратно поддерживают 262 144 цвета. Однако в каждый отдельный момент доступными являются лишь 256 цветов. Драйвер устройства выбирает палитру с равномерным распределением цветов, но иногда этого недостаточно. Палитры позволяют программе набирать до 256 точно задаваемых оттенков.

Если устройство поддерживает палитры GDI, программа может фактически изменить аппаратные регистры, представив в палитре точный набор требуемых ей цветов. Например, если программа должна изображать лесной пейзаж, то ей может понадобиться 150 оттенков зеленого, которые дадут ей возможность передавать нюансы цветов травы, деревьев и т.д. Для того чтобы отобразить пейзаж с лыжниками на фоне покрытых снегом гор, нужно, предположим, 100 оттенков белого, 50 розового и 10 голубого. Палитры дают программе возможность показывать на экране изображения, приближающиеся по качеству к слайдам.

Как и RGB-тройки, индексы палитры по типу - это длинные целые без знака. Единственное различие между ними в том, что в четвертом, неиспользуемом, байте находится флаг, указывающий, что данное значение представляет собой индекс в палитре, а не действительную RGB-тройку сигналов. Макропроцедура PALETTEINDEX позволяет выбрать значение индекса в палитре. Например, чтобы выбрать цвет под номером 137, нужно записать:

```
PALETTEINDEX (137);
```

Третий и последний способ задания цвета связан с заданием RGB-тройки относительно палитры. Как и прямое задание RGB-тройки сигналов, этот способ тоже позволяет задавать красную, зеленую и голубую составляющие цвета. Однако при задании RGB-тройки относительно палитры никогда не происходит смешивания красок. Вместо этого GDI находит в палитре ближайший чистый цвет и использует его. Такой способ задания цвета обеспечивает макропроцедура PALETTEINDEX:

```
PALETTEINDEX (0, 128, 255);
```


Подпрограмма EndPaint

После того как все связанные с сигналом WM_PAINT операции графического вывода завершены, программа вызывает EndPaint, чтобы вернуть контекст устройства дисплея Менеджеру Окон. Когда Менеджер Окон получает контекст устройства, он восстанавливает все его атрибуты в значения по умолчанию, и контекст устройства готов к передаче информации любой программе, которой понадобится нарисовать окно.

Подпрограмма EndPaint также уведомляет Менеджера Окон о том, что поврежденное окно восстановлено. В терминах Windows это означает, что недействительная часть окна стала действительной. Позже в этой главе Вы увидите, что программа может путем обращения к InvalidateRect объявить "клиентную" область окна поврежденной, что приводит к генерации сообщения WM_PAINT. Подпрограмма EndPaint вызывает другую библиотечную подпрограмму Windows ValidateRect, которая приказывает Менеджеру Окон считать данное окно восстановленным.

Если в ответ на WM_PAINT программа не вызовет EndPaint, то Менеджер Окон станет посылать непрерывный поток сообщений WM_PAINT. Пока Менеджер Окон не получит сообщение, что окно восстановлено, он будет просить Вас "починить" окно, а для этого служит сообщение WM_PAINT.

Использование программой пары BeginPaint/EndPaint в ответ на WM_PAINT является ее нормальной реакцией на это сообщение. Вы увидите далее, что использование такого рода пар подпрограмм является обычной практикой программирования в Windows, в частности при записи памяти, чтении файлов и операциях с Clipboard. Подобные конструкции называются сэндвичами Windows. Опишем одну из них подробнее.

Сэндвич Windows

В книгах по структурному программированию термин "сэндвич" не встречается, но для программирования в Windows он играет важную роль.

Подобно своему кулинарному аналогу, сэндвич Windows состоит из трех частей: две наружные (хлеб), между которыми находится третья - начинка. Готовя сэндвич, важно не забыть использовать все три ингредиента и к тому же расположить их в правильном порядке.

На следующем рисунке показан сэндвич программы под Windows. Верхний кусок хлеба - это строка кода, занимающая некоторый ресурс. Нижний кусок хлеба - это последняя строка конструкции, возвращающая ресурс. "Начинка" состоит из одной или более строк кода, использующих ресурс.

Данная конструкция имеет некоторые ресурсы, которые Вы можете использовать наравне с ресурсами других работающих в это время программ. Программы должны взаимодействовать между собой для совместного использования ресурсов и, таким образом, экономить их. Сэндвич - это конструкция, которая специально предназначена для данной цели.

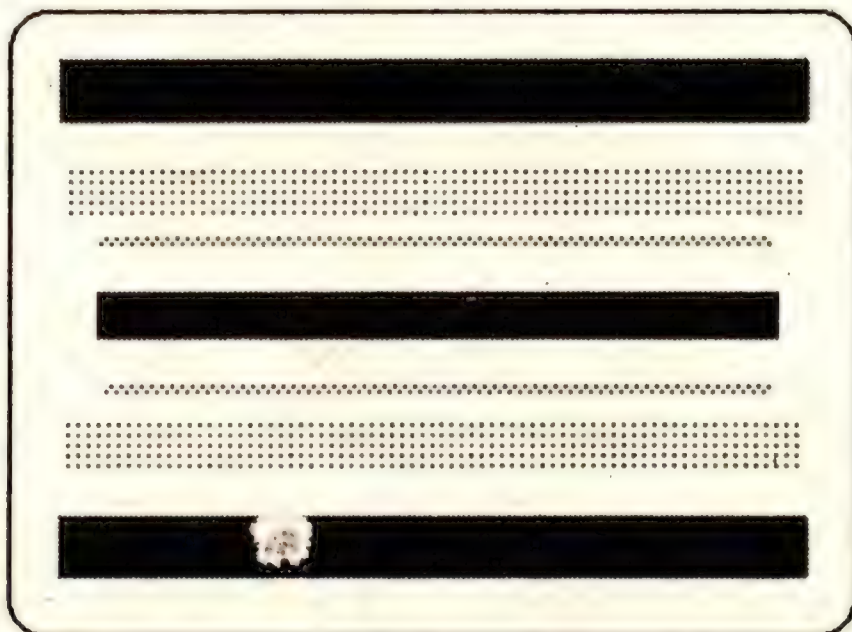


Рис.7.3. Конструкция сэндвича в программе для Windows.

Заимствуемый нами ресурс может быть соединением с устройством графического вывода, дисковым файлом или буфером вырезанного изображения Windows. В целом сэндвич держит ресурс времени, равный длительности действия одного сообщения, как, например, контекст устройства, заимствуемый в ответ на сообщение WM_PAINT:

```
void TSampleWindow::WMPaint(TMessage&)
{
    PAINTSTRUCT ps;
    BeginPaint(HWindow, &ps);    /* Верхний ломтик */
    .
    .                            /* Начинка */
    .
    EndPaint(HWindow, &ps);      /* Нижний ломтик */
}
```

Вызов BeginPaint занимает контекст устройства у Менеджера Окон. Логический номер этого ресурса хранится в переменной hdc. От этой строки и до строки с EndPaint программа полностью распоряжается контекстом устройства. Начинкой сэндвича является то, что наша программа Pixel рисует на экране один пиксель. Когда подпрограмма EndPaint возвращает контекст устройства, больше мы доступа к этому контексту устройства не имеем. После "второго ломтика сэндвича" программа не должна пытаться рисовать в окне.

Необязательно быть поваром в первоклассном ресторане, чтобы сказать, что эти сэндвичи приготовлены неправильно. В первом из них начинка находится вокруг ломтика хлеба, и вы испачкаете пальцы:

```
/* Так делать нельзя: начинка останется на пальцах */
void TSampleWindow::WMPaint(TMessage&)
{
    PAINTSTRUCT ps;
    BeginPaint(HWindow, &ps);    /* Верхний ломтик */
    EndPaint(HWindow, &ps);      /* Нижний ломтик */
    SetPixel (ps.hdc, x, y, RGB (0, 0, 0)); /* Начинка */
}
```

Следующий сэндвич имеет только верхний ломтик хлеба и не имеет нижнего. Опять начинка прилипнет к пальцам.

```
/* Так делать нельзя: всего один ломтик хлеба */
void TSampleWindow::WMPaint(TMessage&)
{
    PAINTSTRUCT ps;

    BeginPaint(HWindow, &ps);    /* Верхний ломтик */
    SetPixel (ps.hdc, x, y, RGB (0, 0, 0)); /* Начинка */
}
```

Этот сэндвич вообще сделан без хлеба, одна начинка.

```
/* Так делать нельзя: вообще без хлеба */
void TSampleWindow::WMPaint(TMessage&)
{
    PAINTSTRUCT ps;
    SetPixel (ps.hdc, x, y, RGB (0, 0, 0)); /* Начинка */
}
```


В некоторых сэндвичевых конструкциях обрабатывается несколько сообщений. Такие сэндвичи можно назвать двухпалубными, или же Дагвуд-сэндвичами (в честь персонажа мультфильма, который любил готовить огромные сэндвичи). Приведем пример, связанный с захватом указателя мыши. Иногда программе бывает удобно запросить исключительный доступ к указателю мыши. Ниже показан фрагмент с примером такого сэндвича:

```
void TSampleWindow::WMLButtonDown(TMessage& Msg)
{
    SetCapture(HWindow);          // Верхний ломтик
    .
    .
}

void TSampleWindow::WMMouseMove(TMessage& Msg)
{
    .
    // Начинка
}

void TSampleWindow::WMLButtonUp(TMessage& Msg)
{
    .
    ReleaseCapture();             // Нижний ломтик
}
```

Двухпалубные сэндвичи мы рассмотрим в главе 16, в которой пойдет речь о вводе с мыши. Перейдем к следующему графическому примитиву. Он не является действительным примитивом GDI; однако мы собираемся "надстроить" его над подпрограммой SetPixel.

Создание маркеров

Наша вторая программа GDI вводит тип графического примитива, не являющегося "родным" примитивом GDI, это маркеры. Наиболее естественная область применения маркеров - графика (рис.7.4). Каждый знак "+" является маркером и отмечает уровень населения за данный год.

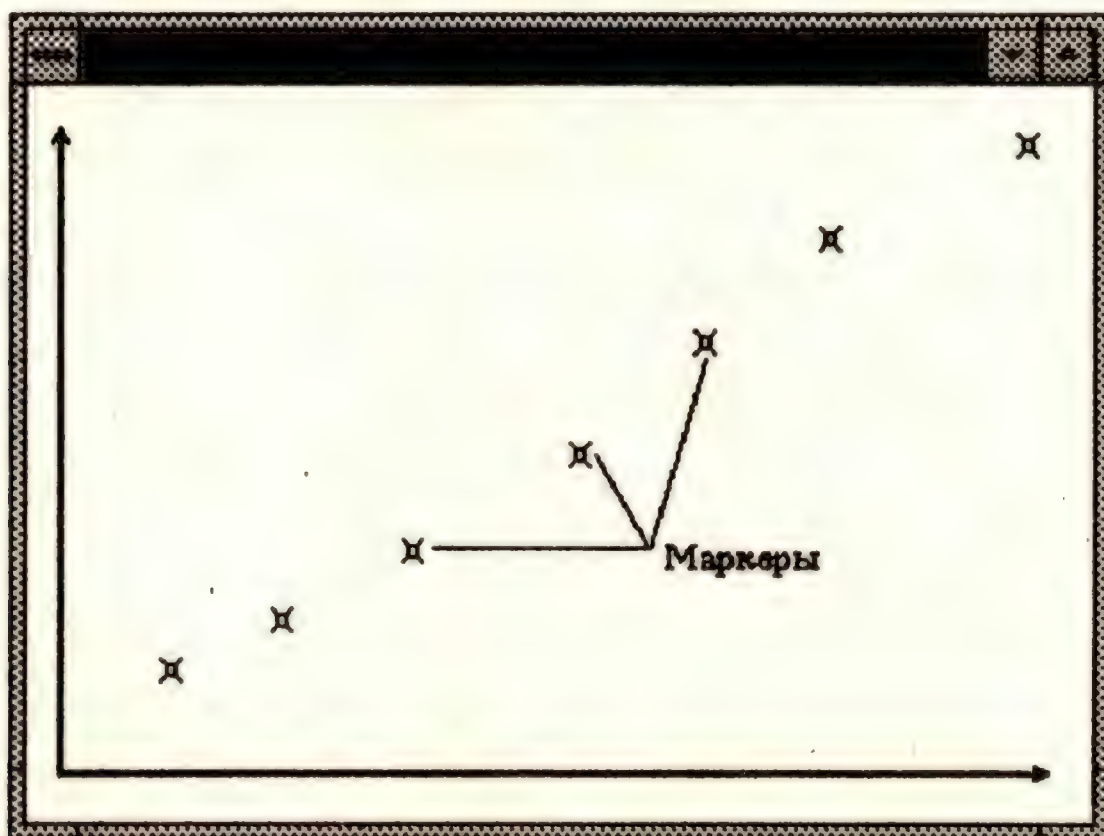


Рис.7.4. Маркеры.

На первый взгляд Вам может показаться странным, зачем нужно выделять маркеры в специальный примитив. Ведь даже из графика видно, что GDI позволяет вывести текст. В чем разница между маркером и текстовым символом?

Как станет ясно из главы 10, подпрограммы текстового вывода GDI упрощают работу с различными размерами и стилями текстов на разных устройствах. Однако эта же гибкость ведет к тому, что нельзя с уверенностью сказать, будет ли данная буква располагаться симметрично относительно задаваемой при ее размещении точки привязки.

Например, используя букву GDI для обозначения на карте точки, где закопаны сокровища, Вы тем самым вносите погрешность. Дело в том, что, например, положение перекрестия буквы X зависит от шрифта.

Маркеры решают эту проблему путем точного центрирования относительно точки привязки, вот почему они и используются в графиках. В следующей главе с помощью маркеров мы будем исследовать поведение различных программ графического вывода.

Один из способов отметить позицию состоит в использовании описанной выше подпрограммы SetPixel. Однако один-единственный пиксель часто трудно заметить на экране, поэтому мы создаем собственный маркер в виде знака (+), сгруппировав в него несколько пикселей.

Представляя другие программы, мы будем использовать маркеры для указания взаимосвязи между координатами, задаваемыми при вызове графического примитива, и реально получившимися. Тем самым мы попробуем прояснить для Вас подход с включением/исключением точек при рисовании линий, закрашенных фигур и при выводе текстов.

На рис.7.5 показан вывод на экран, выполненный нашей программой MARKER.CPP. Это программа помещает маркеры на экран по команде с мыши.



Рис.7.5.Результат работы программы Marker.

MAKEFILE.MAK

.AUTODEPEND

Определения транслятора

INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;C:\BORLANDC\INCLUDE

CC = bcc -c -D_CLASSDLL -H -ml -WS -w -I\$(INC)

Неявные правила

.c.obj:
\$(CC) {\$< }

.cpp.obj:
\$(CC) {\$< }

Явные правила

Marker.exe: Marker.res Marker.def Marker.obj
tlink /c/C/n/P-/Twe/x @Marker.LNK
RC Marker.res Marker.exe

Зависимости отдельных файлов

Marker.obj: Marker.cpp


```
Marker.res: Marker.rc Marker.cur Marker.ico
RC -R -FO Marker.res Marker.RC
```

MARKER.LNK

```
c:\borlandc\lib\c0wl.obj+
Marker.obj
Marker,Marker
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwl.lib+
import.lib+
mathl.lib+
cl.lib
Marker.def
```

MARKER.CPP

```
/*
Marker.CPP Прорисовка маркера в ответ на нажатие клавиши мыши
*/
#include <owl.h>
#define MOUSEX(arg) (arg.LP.Lo)
#define MOUSEY(arg) (arg.LP.Hi)

/*
Константы
*/
const int MAXPOINTS = 32;
const int MARKERSIZE = 3;

/*
Определения классов
*/
class TMarkerApplication : public TApplication
{
public:
    TMarkerApplication (LPSTR lpszName, HANDLE hInstance,
                        HANDLE hPrevInstance, LPSTR lpszCmdline, int nCmdShow);
    virtual void InitMainWindow ();
};

class TMarkerWindow : public TWindow
{
public:
    TMarkerWindow (PTWindowsObject pParent, LPSTR lpszTitle,
                  PTModule pmModule);
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WNDCLASS&);
    virtual void WMLButtonDown(TMessage& Msg) = [WM_LBUTTONDOWN];
    virtual void WMPaint(TMessage& Msg) = [WM_PAINT];
    int cpt;
    POINT pt[MAXPOINTS];
};

/*
Прототипы функций
*/
void DrawMarker (HDC hDC, int x, int y);
```



```

/*
Основная функция: WinMain
*/
int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
{
    TMarkerApplication Marker ("Marker", hInstance,
                               hPrevInstance, lpszCmdLine, nCmdShow);
    Marker.Run();
    return Marker.Status;
}

/*
Компонентная функция класса прикладной программы
*/
TMarkerApplication::TMarkerApplication (LPSTR lpszName,
                                         HANDLE hInstance, HANDLE hPrevInstance,
                                         LPSTR lpszCmdLine, int nCmdShow)
    :TApplication (lpszName, hInstance,
                  hPrevInstance, lpszCmdLine, nCmdShow)
{
    /* Здесь инициализируется прикладная программа */
}

/*
Компонентная функция класса прикладной программы
*/
void TMarkerApplication::InitMainWindow ()
{
    MainWindow = new TMarkerWindow (NULL, "Marker", NULL);
}

/*
Компонентная функция класса TMarkerWindow
*/
TMarkerWindow::TMarkerWindow (PTWindowsObject pwParent,
                              LPSTR lpszTitle, PTModule pmModule)
    :TWindow (pwParent, lpszTitle, pmModule)
{
    cpt=10;
    for(int i=0; i<cpt; i++)
    {
        pt[i].x=i*10;
        pt[i].y=i*i;
    }
}

/*
Компонентная функция класса TMarkerWindow
*/
LPSTR TMarkerWindow::GetClassName ()
{
    return "Marker:MAIN";
}

```



```

/*
Компонентная функция класса TMarkerWindo
*/
void TMarkerWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
}

/*
Компонентная функция класса TMarkerWindo
*/
void TMarkerWindow::WMLButtonDown (TMessage& Msg)
{
    if (cpt < MAXPOINTS)
    {
        pt[cpt].x = MOUSEX(Msg);
        pt[cpt].y = MOUSEY(Msg);
        InvalidateRect (HWindow, NULL, TRUE);
        cpt++;
    }
}

/*
Компонентная функция класса TMarkerWindo
*/
void TMarkerWindow::WMPaint(TMessage&)
{
    int i;
    PAINTSTRUCT ps;

    BeginPaint (HWindow, &ps);
    for (i=0; i<cpt; i++) DrawMarker (ps.hdc, pt[i].x, pt[i].y);
    EndPaint(HWindow, &ps);
}

/*
Функция определения маркера
*/
void DrawMarker (HDC hdc, int x, int y)
{
    DWORD dwColor;
    int i;

    dwColor = GetPixel (hdc, x, y);
    dwColor = ~dwColor;
    SetPixel (hdc, x, y, dwColor);
    for (i=1; i< MARKERSIZE; i++)
    {
        dwColor = GetPixel (hdc, x+i, y);
        dwColor = ~dwColor;
        SetPixel (hdc, x+i, y, dwColor);

        dwColor = GetPixel (hdc, x-i, y);
        dwColor = ~dwColor;
        SetPixel (hdc, x-i, y, dwColor);
    }
}

```



```

        dwColor = GetPixel (hdc, x, y+i);
        dwColor = ~dwColor;
        SetPixel (hdc, x, y+i, dwColor);

        dwColor = GetPixel (hdc, x, y-i);
        dwColor = ~dwColor;
        SetPixel (hdc, x, y-i, dwColor);
    }
}

```

MARKER.RC

```

snapshot icon Marker.ico
hand cursor Marker.cur

```

MARKER.DEF

```

NAME MARKER
EXETYPE WINDOWS
DESCRIPTION 'Marker - Marker demo.
CODE MOVEABLE DISCARDABLE
DATA MOVEABLE MULTIPLE

HEAPSIZE 512
STACKSIZE 5120

```

Программа Marker содержит подкласс стандартного оконного класса TWindow, обрабатывающий два сообщения: WM_LBUTTONDOWN и WM_PAINT. Как и в программах MIN и Pixel, класс TWindow выполняет все необходимые каждой программе для Windows действия по инициализации и закрытию.

Сообщение WM_LBUTTONDOWN - это сообщение, связанное с мышью. Для обработки его Marker создает функцию - обработчик сообщения WM_LBUTTONDOWN. Сообщение WM_LBUTTONDOWN направляется, когда одновременно курсор мыши находится в "клиентной" области и пользователь нажал левую кнопку мыши. (Вы можете получить это сообщение и вне "клиентной" области, если мышь была захвачена. Подробности об этом см. в главе 16.)

Всякий раз при получении нашим оконным объектом данного сообщения от мыши он записывает координаты курсора мыши X и Y. Для всех сообщений мыши позиция курсора находится в параметре LParam и выражается в координатах клиентной области. Вспомните, что эта система координат имеет начало (0,0) в верхнем левом углу клиентной области.

Оконный объект хранит позиции курсора мыши X и Y в массиве структур POINT. Тип данных POINT определен в WINDOWS.H как:

```

typedef struct tagPOINT
{
    int x;
    int y;
} POINT;

```

Программа Marker использует две макропроцедуры MOUSEX и MOUSEY, позволяющие получить из LParam координаты мыши X и Y. Эти макропроцедуры определены в MARKER.CPP так:

```

#define MOUSEX(arg) (arg.LP.Lo)
#define MOUSEY(arg) (arg.LP.Hi)

```

Основное их достоинство с точки зрения доступа к параметрам сообщений состоит в том, что они помогут Вам переносить программы для Windows 3.0 в другие опе-

рационные системы, поддерживающие интерфейс прикладных программ Windows. В оставшейся части книги Вы увидите, что они используются всякий раз, когда нужно принять значение параметра сообщения.

После того как новая точка добавлена в массив, оконный объект вызывает перерисовку окна:

```
InvalidateRect (HWindow, NULL, TRUE);
```

Эта подпрограмма вызывает генерацию сообщения WM_PAINT, говоря Windows, что "окно полностью разбито".

Подпрограмма InvalidateRect принимает три параметра:

- * логический номер перерисовываемого окна;
- * дальний указатель на объявленный поврежденным прямоугольник окна;
- * флаг "сначала стереть" (значение TRUE вызывает стирание фона перед перерисовкой, FALSE отменяет стирание).

В ответ на сообщение WM_PAINT наша программа Marker с помощью сэндвичевой конструкции BeginPaint/EndPaint занимает у Менеджера Окон контекст устройства. Начинкой сэндвича является цикл DrawMarker, который вызывает нашу подпрограмму рисования маркера, задавая положение X и Y каждой точки в массиве pt. Подпрограмма рисования маркера DrawMarker построена с помощью двух подпрограмм GDI GetPixel и SetPixel. Как правило, если Вы не нашли подпрограмму GDI, выполняющую все, что Вам нужно, легче всего написать свою подпрограмму, которая будет являться надстройкой над существующими подпрограммами GDI, как, например, в данном случае.

Наша предыдущая программа Pixel рисовала единственный черный пиксель, вызывая SetPixel с набором параметров:

```
SetPixel (hdc, r.right/2, r.bottom/2, RGB (0,0,0));
```

Мы могли бы создавать маркер, просто несколько раз делая подобный вызов с разными координатами. Однако, что произойдет, если мы попытаемся в этом режиме нарисовать маркер на черном фоне? Он попросту исчезнет.

Наша подпрограмма решает эту проблему за счет алгебры логики. Прежде чем записывать пиксель, мы прочитываем его с помощью GetPixel. Последняя возвращает RGB-тройку, которая говорит нам о цвете пикселя. Затем мы выполняем поразрядное отрицание RGB-значений, чтобы инвертировать цвет, и записываем пиксель, используя SetPixel. Вот фрагмент кода, о котором шла речь:

```
dwColor = GetPixel (hdc, x, y);  
dwColor = ~dwColor;  
SetPixel (hdc, x, y, dwColor);
```

Сделав последовательность такого рода вызовов, мы нарисует маркер, видимый на любом фоне, независимо от его первоначального цвета.

Операция "~" выполняет поразрядную логическую операцию дополнения операнда, которым в данном случае является значение RGB-тройки, до единицы. Все нули станут единицами, а все единицы - нулями. Как это ни странно, алгебра логики играет важную роль в мире графики. Далее мы увидим, как режимы графического вывода позволяют нам выполнять логические операции над рисуемыми линиями, закрашенными фигурами и копируемыми битовыми матрицами.

Теперь Вы убедились, что данный способ позволяет эффективно создавать маркеры на экране дисплея. Однако он имеет одно ограничение: два маркера, будучи совмещенными, отрицают друг друга. Они исчезают из видимости. Как быть?

Один из способов состоит в том, чтобы программа анализировала во время рисования список точек. Дублирующая точка просто игнорируется. Другой способ состоит в том, чтобы такой анализ выполнялся всякий раз при нажатии пользователем

кнопки мыши. Если точка уже отмечена маркером, программа может с помощью сигнала сообщить об ошибке.

Мы представили Вам подпрограмму рисования маркеров, чтобы продемонстрировать на ее примере, как работают "родные" подпрограммы GDI. Теперь воспользуемся этой подпрограммой при изучении графических возможностей GDI. Это основной вопрос следующей главы.

Глава 8. Рисование линий

Прежде чем начать учиться рисовать линии с помощью подпрограмм GDI, остановимся на минуту и подумаем: а что же такое линия? Линия - это геометрическая фигура, создаваемая множеством точек. Линии называют незамкнутыми фигурами, имея в виду, что нас интересует сама линия, а не ограничиваемая ею область. В главе 9 мы рассмотрим замкнутые фигуры, в которых линия в совокупности с ограничиваемой ею областью образует единую геометрическую фигуру.

Линия GDI представляет собой геометрическую фигуру, создаваемую подпрограммой рисования линий GDI. Каждая такая линия имеет начальную и конечную точки. GDI рисует линии с использованием алгоритма включения/исключения, т.е. начальная точка включается в линию, а конечная исключается из нее.

Утилита Magnify позволит Вам убедиться в этом (рис.8.1). Magnify растягивает пиксели на экране, так что они становятся явно видны. (Исходный код Magnify приводится в Приложении Н.) На создаваемом этой утилитой рисунке благодаря маркерам начальной и конечной точек явно видна работа алгоритма включения/исключения.

Белый пиксель в перекрестии маркера говорит о том, что начальная точка включена в линию. Маркер рисуется с инверсией пикселей. В конечной точке пиксель в перекрестии маркера черный. Это означает, что она исключена из линии.

Хотя такой алгоритм рисования линий на первый взгляд может показаться необычным, он позволяет рисовать сложные фигуры с помощью множества простых линий. В главе 16 мы рассмотрим создание объектов, которые можно перемещать по экрану с помощью мыши, там это свойство особенно важно при использовании различных режимов графического вывода.

Все подпрограммы GDI для рисования линий используют алгоритм включения/исключения. Рассмотрим эти подпрограммы, чтобы ознакомиться с возможностями GDI. При этом мы остановимся на атрибутах контекста устройства, влияющих на отображение линий.

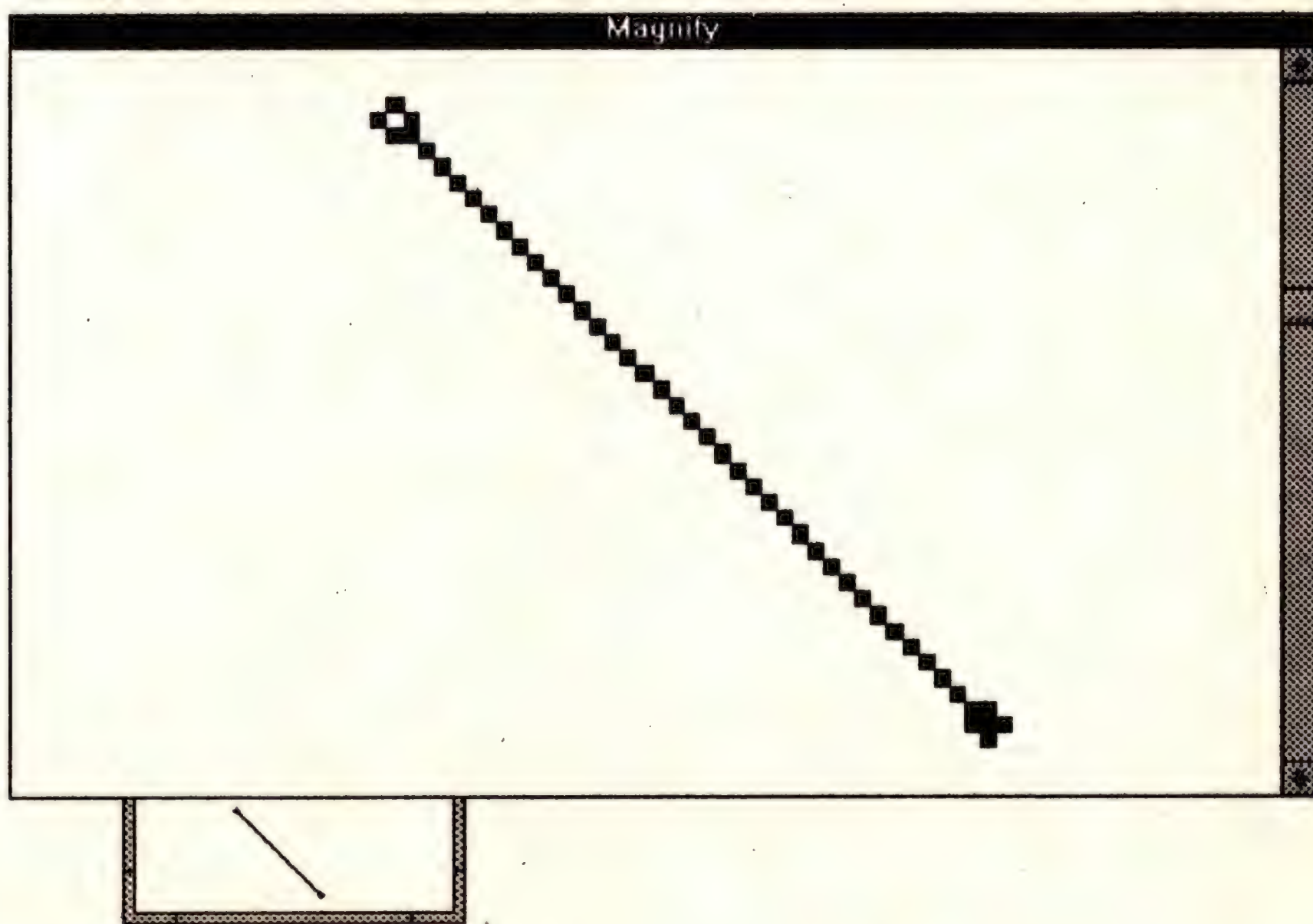


Рис.8.1. Прорисовка линии с помощью программы Magnify.

Примитивы для рисования линий

GDI имеет четыре подпрограммы для рисования линий: MoveTo, LineTo, PolyLine и Arc. Рассмотрим их поочередно. Начнем с двух первых, которые всегда используются в паре: MoveTo и LineTo.

Подпрограммы MoveTo и LineTo

Первая из этих подпрограмм, MoveTo, фактически линию не рисует. Она просто помещает пару X- и Y-координат в атрибут контекста устройства, который называется текущей позицией. Вторая подпрограмма, LineTo, использует это значение как начальную точку при рисовании линии. Функция LineTo сама принимает вторую точку как параметр. Вот как выглядят эти подпрограммы для рисования линии из X1,Y1 в X2,Y2:

```
MoveTo (hdc, X1, Y1);  
LineTo (hdc, X2, Y2);
```

После того как LineTo нарисовала линию, она обновляет текущую позицию в контексте устройства, помещая туда адрес конечной точки линии. Используя показанную ниже последовательность вызовов, Вы можете соединить линиями любой набор точек:

```
MoveTo (hdc, X1, Y1);  
LineTo (hdc, X2, Y2);  
LineTo (hdc, X3, Y3);  
LineTo (hdc, X4, Y4);
```

При таком способе вызова эти подпрограммы дают тот же эффект, что может быть получен с помощью подпрограммы Polyline. Вас может удивить эта избыточность GDI.

Однако это всего лишь вопрос удобства. Может оказаться, что парой MoveTo/LineTo пользоваться легче, так как Polyline требует начального создания массива координат X и Y точек.

Вместе с тем, несмотря на специальный формат параметров, Polyline выигрывает в быстродействии. Это прямой результат того, что при лишнем вызове функций расходуется лишнее время. Один вызов Polyline позволяет нарисовать ломаную, в отличие от множества вызовов для достижения того же эффекта при использовании пары MoveTo/LineTo.

Как и пара MoveTo/LineTo, Polyline рисует прямые линии. Однако она не работает с текущей точкой контекста устройства, а использует исключительно передаваемый ей в параметре массив точек. Если мы поместили точки (X1,Y1), (X2,Y2), (X3,Y3) и (X4,Y4) в массив, например:

```
POINTS pt[] = {X1, Y1, X2, Y2, X3, Y3, X4, Y4};
```

то следующий вызов Polyline соединит эти точки:

```
Polyline (hdc, pt, 4);
```

Функция Arc рисует дуги эллипса (рис.8.2). Параметры, передаваемые в Arc, определяют три границы: прямоугольник, ограничивающий фигуру, если бы она была полной, начальную точку и конечную. Если начальная и конечная точки совпадают, то Arc рисует полную фигуру (эллипс или окружность, если ограничивающий прямоугольник является квадратом). В противном случае рисуется часть эллипса.

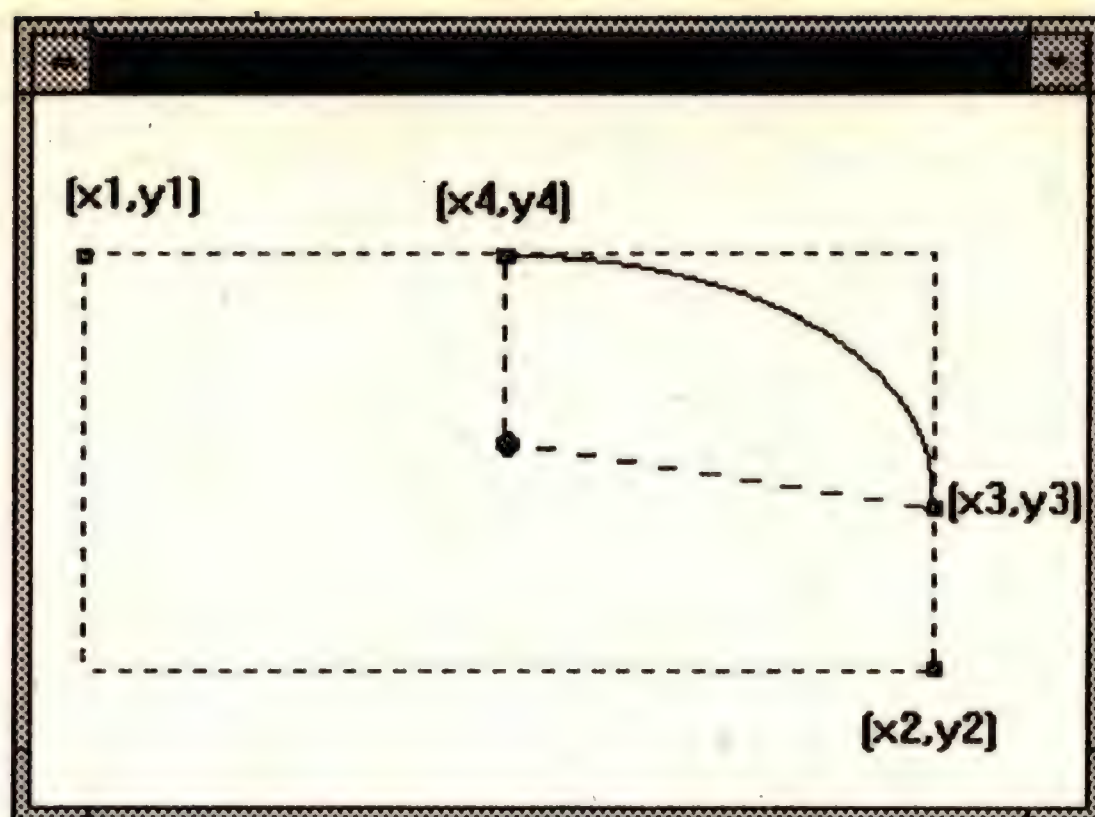


Рис.8.2. Функция Arc.

Как и для предыдущих подпрограмм, здесь тоже важную роль играет отношение между начальной и конечной точками, поскольку GDI использует алгоритм включения/исключения. Кроме того, дуги рисуются в направлении против часовой стрелки (рис.8.3). Если начальную и конечную точки дуги поменять местами, то дуга получится совсем другой.

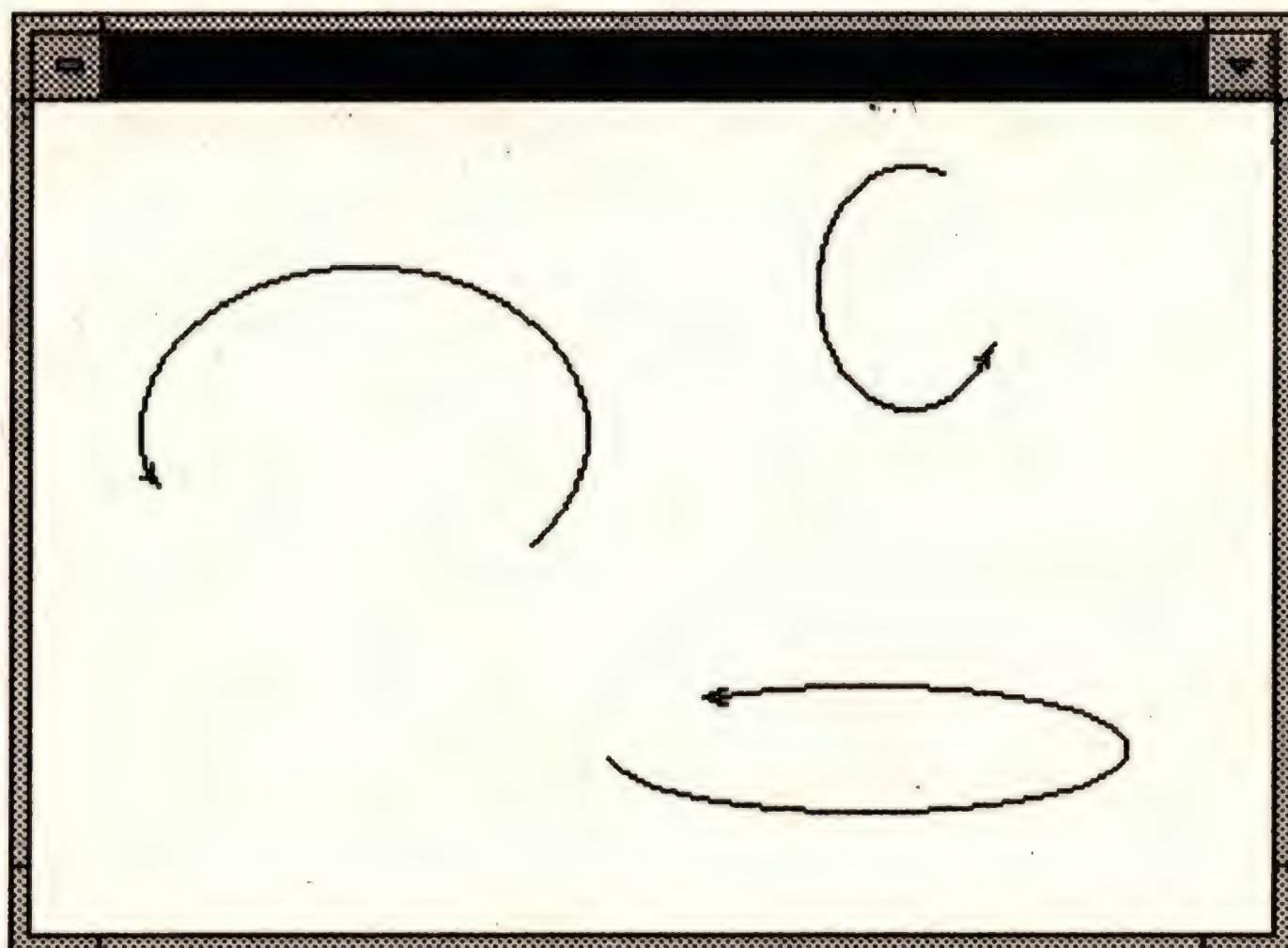


Рис.8.3. Дуги (нарисованные в направлении против часовой стрелки).

Наш пример программы может использовать все три типа примитивов для рисования линий. Вы вводите точки нажатием кнопки мыши в "клиентной" области. Первые четыре точки используются для рисования линий с помощью пары MoveTo/LineTo. Вторые четыре - для рисования полилинии с помощью Polyline. И, наконец, четыре последние служат входом в подпрограмму Arc. Полный листинг программы Lines приводится ниже.

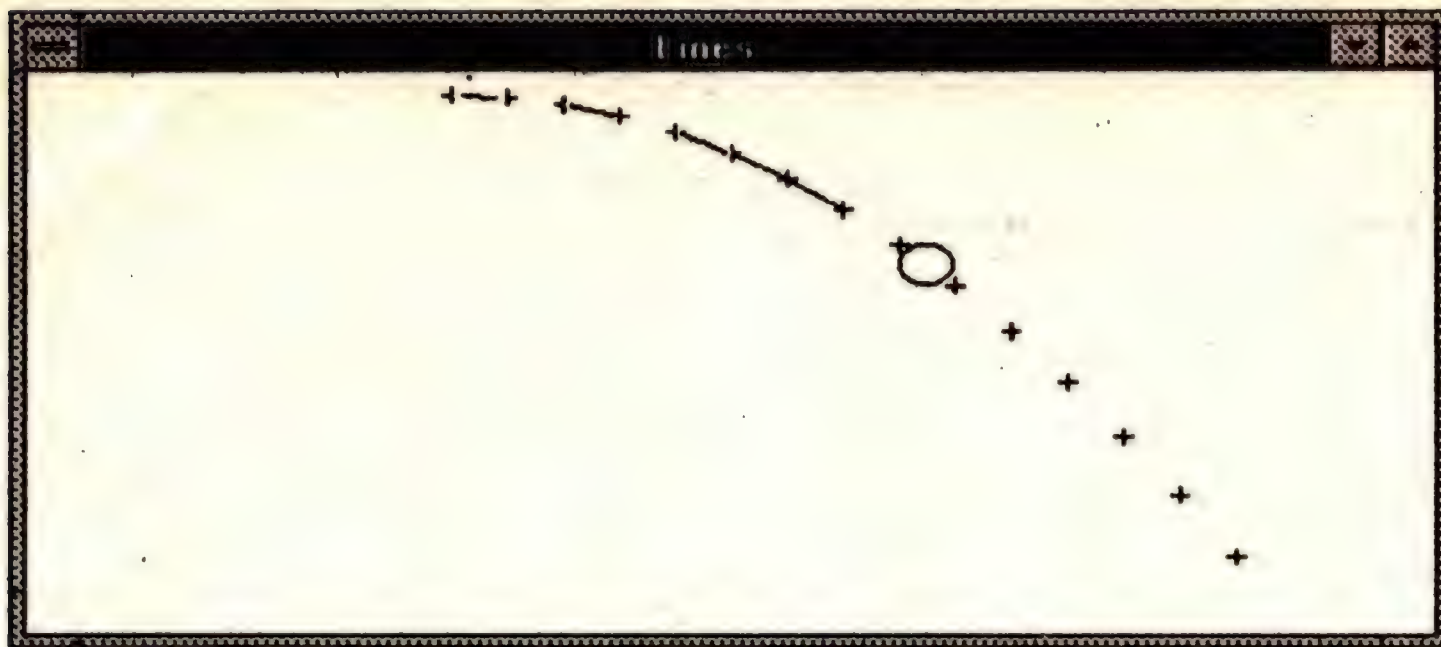


Рис.8.4. Три графических примитива, которые рисуются с помощью программы Lines.

MAKEFILE.MAK

.AUTODEPEND

```
# Определения компилятора
INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;C:\BORLANDC\INCLUDE
CC = bcc -c -D_CLASSDLL -H -ml -WS -w -IS(INC)

# Неявные правила
.c.obj:
    $(CC) {$<}

.cpp.obj:
    $(CC) {$<}

# Явные правила
Lines.exe: Lines.res Lines.def Lines.obj
    tlink /c/C/n/P-/Twe/x @Lines.LNK
    RC Lines.res Lines.exe

# Зависимости
Lines.obj: Lines.cpp
Lines.res: Lines.rc Lines.cur Lines.ico
    RC -R -FO Lines.res Lines.RC
```

LINES.LNK

```
c:\borlandc\lib\c0wl.obj+
Lines.obj
Lines,Lines
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwl.lib+
import.lib+
mathl.lib+
cl.lib
Lines.def
```



```

/*
LINES.CPP Рисование линий с использованием функций
MoveTo/LineTo, Polyline и Arc
*/
#include <owl.h>

#define MOUSEX(arg) (arg.LP.Lo)
#define MOUSEY(arg) (arg.LP.Hi)

/*
Константы
*/
const int MAXPOINTS = 12;
const int MARKERSIZE = 3;

/*
Объявление классов
*/
class TLinesApplication : public TApplication
{
public:
    TLinesApplication (LPSTR lpszName, HANDLE hInstance,
                       HANDLE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow);
    virtual void InitMainWindow ();
};

class TLinesWindow : public TWindow
{
public:
    TLinesWindow (PTWindowsObject pwParent, LPSTR lpszTitle,
                  PTModule pmModule);
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WNDCLASS&);
    virtual void WMLButtonDown(TMessage& Msg) =
        [WM_FIRST+WM_LBUTTONDOWN];
    virtual void WMPaint(TMessage& Msg) = [WM_FIRST + WM_PAINT];
private:
    int cpt;
    POINT pt[MAXPOINTS];
};

/*
Прототипы функций
*/
void DrawMarker (HDC hDC, int x, int y);

/*
Главная функция: WinMain
*/
int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
{
    TLinesApplication Lines ("Lines", hInstance, hPrevInstance,
                             lpszCmdLine, nCmdShow);
    Lines.Run();
    return Lines.Status;
}

```



```

/*
Компонентная функция класса прикладных программ
*/
TLinesApplication::TLinesApplication (LPSTR lpszName,
                                     HANDLE hInstance, HANDLE hPrevInstance,
                                     LPSTR lpszCmdLine, int nCmdShow)
: TApplication (lpszName, hInstance,
               hPrevInstance, lpszCmdLine, nCmdShow)
{
    /* Здесь инициализируется прикладная программа */
}

/*
Компонентная функция класса прикладных программ
*/
void TLinesApplication::InitMainWindow ()
{
    MainWindow = new TLinesWindow (NULL, "Lines", NULL);
}

/*
Компонентная функция класса TLinesWindow
*/
TLinesWindow::TLinesWindow (PTWindowsObject pwParent,
                           LPSTR lpszTitle, PTModule pmModule)
: TWindow (pwParent, lpszTitle, pmModule)
{
    cpt = 15;
    for(int i=0; i<cpt; i++)
    {
        pt[i].x=150+i*20;
        pt[i].y=10+i*i;
    }
}

/*
Компонентная функция класса TLinesWindow
*/
LPSTR TLinesWindow::GetClassName ()
{
    return "Lines:MAIN";
}

/*
Компонентная функция класса TLinesWindow
*/
void TLinesWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
}

/*
Компонентная функция класса TLinesWindow
*/
void TLinesWindow::WMLButtonDown (TMessage& Msg)
{

```



```

    if (cpt < MAXPOINTS)
    {
        pt[cpt].x = MOUSEX(Msg);
        pt[cpt].y = MOUSEY(Msg);
        InvalidateRect (HWindow, NULL, TRUE);
        cpt++;
    }
}

/*
Компонентная функция класса TLinesWindow
*/
void TLinesWindow::WMPaint(TMessage&)
{
    int i;
    PAINTSTRUCT ps;
    BeginPaint (HWindow, &ps);

    /*
    Рисование линий с помощью функций MoveTo / LineTo.
    */
    if (cpt > 1)
    {
        MoveTo (ps.hdc, pt[0].x, pt[0].y);
        LineTo (ps.hdc, pt[1].x, pt[1].y);
    }
    if (cpt > 3)
    {
        MoveTo (ps.hdc, pt[2].x, pt[2].y);
        LineTo (ps.hdc, pt[3].x, pt[3].y);
    }

    /*
    Рисование линий с помощью Polyline.
    */
    if (cpt > 4) Polyline (ps.hdc, &pt[4], 4);

    /*
    Рисование дуги.
    */
    if (cpt >= 12)
        Arc (ps.hdc, pt[8].x, pt[8].y,
            pt[9].x, pt[9].y,
            pt[10].x, pt[10].y,
            pt[11].x, pt[11].y);

    /*
    Подсвечивание контрольных точек маркерами.
    */
    for (i=0; i<cpt; i++)
        DrawMarker (ps.hdc, pt[i].x, pt[i].y);
    EndPaint(HWindow, &ps);
}

/*
Функция прорисовки маркера
*/
void DrawMarker (HDC hdc, int x, int y)
{
    DWORD dwColor;

```



```

int i;
dwColor = GetPixel (hdc, x, y);
dwColor = ~dwColor;
SetPixel (hdc, x, y, dwColor);
for (i=1;i<=MARKERSIZE;i++)
{
    dwColor = GetPixel (hdc, x+i, y);
    dwColor = ~dwColor;
    SetPixel (hdc, x+i, y, dwColor);

    dwColor = GetPixel (hdc, x-i, y);
    dwColor = ~dwColor;
    SetPixel (hdc, x-i, y, dwColor);

    dwColor = GetPixel (hdc, x, y+i);
    dwColor = ~dwColor;
    SetPixel (hdc, x, y+i, dwColor);

    dwColor = GetPixel (hdc, x, y-i);
    dwColor = ~dwColor;
    SetPixel (hdc, x, y-i, dwColor);
}
}

```

LINES.RC

```

snapshot icon Lines.ico
hand cursor Lines.cur

```

LINES.DEF

```

NAME LINES
EXETYPE WINDOWS
DESCRIPTION 'Lines -- Sample Line Drawing Program'
CODE MOVEABLE DISCARDABLE
DATA MOVEABLE MULTIPLE
HEAPSIZE 512
STACKSIZE 5120

```

Теперь, изучив подпрограммы GDI, предназначенные для рисования линий, рассмотрим, какие атрибуты контекста устройства при этом задействуются.

Атрибуты контекста устройства

Для рисования линий GDI используется пять атрибутов контекста устройства:

Атрибут графического вывода	Комментарий
Цвет фона	Второй цвет для прерывистых перьев
Режим фона	Включает/выключает цвет фона
Текущая позиция	(X,Y)-позиция для подпрограммы LineTo
Режим графического вывода	Логическая операция при рисовании
Перо	Цвет, толщина и стиль линии

Несомненно, самым важным атрибутом является перо, определяющее, как будет выглядеть линия: ее цвет, ширина и стиль (или шаблон, например, сплошная линия,

пунктир и т.п.). Термин "линия со стилем" часто означает линию, отличную от сплошной: точечную, штриховую и т.п.

Второй по значимости атрибут - режим графического вывода, в котором можно задать логическую операцию, чтобы применять ее при выводе между новым и существующим пикселями. Об этом чуть позже.

Из остальных трех атрибутов два влияют на линии со стилем (прерывистые линии), это цвет и режим фона. GDI использует цвет фона для закрашивания разрывов в линии, т.е. между штрихами и пунктирами в штрих-пунктирной линии. Режим фона определяет, будет ли происходить закрашивание разрывов линии или нет. Используя эти атрибуты, помните, что они также влияют и на процесс рисования закрашенных фигур (при использовании штрихующей кисти) и текстов.

Цвет фона устанавливается с помощью подпрограммы `SetBkColor`, определяемой как:

`SetBkColor (hDC, crColor)`

- * `hDC` - логический номер контекста устройства.
- * `crColor` - относительный номер цвета; это RGB-тройка, индекс палитры или RGB-тройка относительно палитры.

Следующая строка устанавливает голубой цвет фона:

`SetBkColor (hDC, RGB(0, 0, 0xEF));`

Для установки режима отображения фона служит подпрограмма `SetBkMode`, определяемая как:

`SetBkMode (hDC, nBkMode)`

- * `hDC` - логический номер контекста устройства;
- * `nBkMode` работает как ключ вкл/выкл. Для включения отображения фона нужно задать значение `OPAQUE`, а для выключения - `TRANSPARENT`.

Последний атрибут контекста устройства, текущую позицию, мы рассматривали в связи с подпрограммами `MoveTo/LineTo`. Это X,Y-значение, генерируемое подпрограммами во время рисования, `MoveTo` устанавливает текущую позицию, а `LineTo` использует ее в качестве начальной точки линии. Завершив линию, `LineTo` корректирует текущую позицию, делая ее равной конечной точке линии.

Рассмотрим поближе перья, а также способ их создания и управления ими.

О перьях

Перо представляет собой графический атрибут контекста устройства, характеризующий способ проведения линий. Перья характеризуются тремя параметрами: цветом, шириной и стилем. Если хотите, то можете каждый из этих параметров считать отдельным графическим атрибутом. В графическом программировании, когда атрибуты группируются таким способом, их называют узлом атрибутов. Они удобны тем, что позволяют сразу устанавливать несколько атрибутов.

GDI дает Вам возможность использования перьев. Перья могут использоваться как программами, так и устройствами. GDI имеет несколько общих перьев, которыми может воспользоваться любая программа. Кроме того, программа может создать набор собственных перьев и разделить их между различными устройствами, обеспечить тем самым экономию памяти для хранения графических атрибутов. GDI очень осторожно расходует память, так как первые версии Windows должны были обходиться 640 Кбайтами памяти.

Независимость перьев и устройств

Каким образом различные устройства могут использовать перья описывает термин логическое перо. Вообще говоря, перо представляет собой запрос к устройству

на рисование линий определенного вида. Когда подпрограмма GDI готовится к рисованию на конкретном устройстве, она дает ему указание подготовить перо. Именно в этот момент драйвер устройства создает структуры данных, которые необходимы для проведения линий с желаемыми свойствами. Данный аспект скрыт интерфейсом драйверов устройств (GDI), однако именно он обеспечивает одновременное использование перьев различными устройствами.

Создание и использование перьев

При начальной загрузке Windows GDI создает набор перьев, которые могут одновременно использоваться всеми программами. Это общие, или стандартные перья: черное, белое и нулевое (невидимое). Нулевое перо просто резервирует место для настоящего пера, так как контекст устройства должен содержать действительное перо. Остальные два стандартных пера проводят сплошные линии толщиной в один пиксель.

Каждое перо идентифицируется логическим номером. Чтобы получить логический номер стандартного пера, можно воспользоваться подпрограммой `GetStockObject`:

```
HPEN hpen;  
hpen = GetStockObject (BLACK_PEN); /* или */  
hpen = GetStockObject (WHITE_PEN); /* или */  
hpen = GetStockObject (NULL_PEN);
```

Получив логический номер пера, Вы вызываете еще одну подпрограмму, чтобы установить перо в контексте устройства:

```
SelectObject (hdc, hpen);
```

После того как перо установлено в контексте устройства, оно может быть задействовано для последующего рисования линий. Это относится к линиям, получаемым с помощью `MoveTo/LineTo`, `Polyline` и `Arc`.

Нужно помнить, что одновременно в контексте устройства может находиться всего одно перо. Следовательно, взяв в контексте устройства новое перо, Вы автоматически удаляете оттуда старое. Для удобства подпрограмма `SelectObject` возвращает логический номер удаленного пера. Например, в следующем случае контекст устройства не изменится:

```
/* контекст устройства не изменится */  
hpenOld = SelectObject (hdc, hpenNew);  
SelectObject (hdc, hpenOld);
```

Если трех стандартных перьев Вам недостаточно, GDI предоставит две подпрограммы для создания собственных: `CreatePen` и `CreatePenIndirect`. Единственное различие между ними состоит в способе задания параметров. Синтаксис `CreatePen` следующий:

```
hpen = CreatePen (nPenStyle, nWidth, crColor);
```

* `nPenStyle` выбирает стиль пера, устанавливая соответствующий флаг (рис.8.5).

* `nWidth` устанавливает ширину по X в пикселях. Рассматривая преобразования координат, мы увидим, что значение толщины отражает систему координат, установленную в контексте устройства, которому принадлежит данное перо.

* `crColor` - относительный номер цвета; это (a) RGB-тройка, (b) индекс палитры или (c) RGB-тройка относительно палитры.

Итак, для создания черного пера с толщиной в один пиксель нужно записать:

```
hpen = CreatePen (PS_SOLID, 1, RGB (0, 0, 0));
```

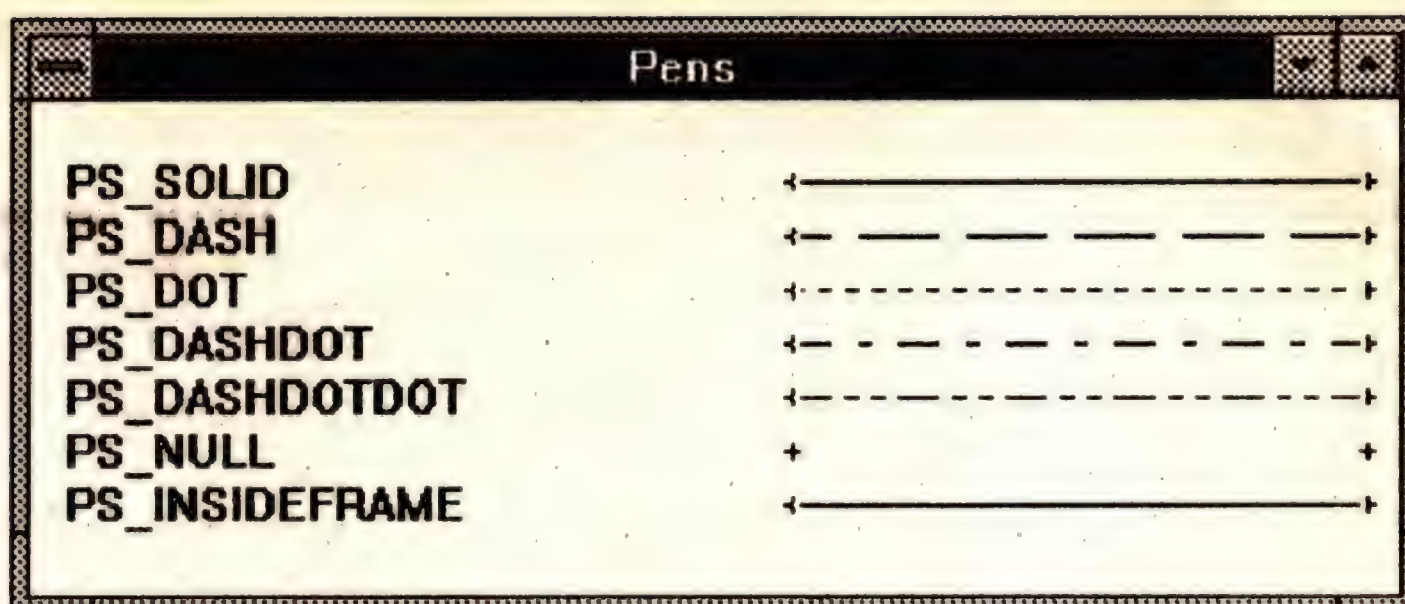



Рис.8.5. Семь стилей, используемых GDI.

Существуют следующие стили пера:

<i>PS_SOLID</i>	сплошная линия
<i>PS_DASH</i>	длинные штрихи
<i>PS_DOT</i>	короткие штрихи
<i>PS_DASHDOT</i>	штрих-пунктир (длинная - короткая - ...)
<i>PS_DASHDOTDOT</i>	двойной штрих-пунктир (длинная - короткая - короткая - ...)
<i>PS_NULL</i>	нулевая линия
<i>PS_INSIDEFRAME</i>	сплошная линия, рисующая по внутренней границе рамки

Синтаксис CreatePenIndirect другой:

```
LOGPEN logpen;
hpen = CreatePenIndirect (&logpen)
```

LOGPEN определяется в WINDOWS.H следующим образом:

```
typedef struct tagLOGPEN
{
    WORD lopnStyle;
    POINT lopnWidth;
    DWORD lopnColor;
} LOGPEN;
```

Одно различие между CreatePen и CreatePenIndirect состоит в том, что структура LOGPEN использует структуру POINT для хранения толщины пера. Вспомните, что в структуру POINT входит два компонента - для X и для Y. Для создания такого же черного пера Вы записываете:

```
LOGPEN logpen;
logpen.lopnStyle = PS_SOLID;
logpen.lopnWidth.x = 1;
logpen.lopnColor = RGB(0, 0, 0);
hpen = CreatePenIndirect (&logpen);
```

GDI поддерживает семь стилей пера. Последний из них, PS_INSIDEFRAME, дает тот же результат, что и стиль PS_SOLID, но с двумя существенными различиями. Это единственный стиль, допускающий смешанные цвета. Все прочие стили должны иметь чистый цвет.

В контексте работы с закрашиваемыми фигурами стиль PS_INSIDEFRAME обладает рядом уникальных свойств. Перо, имеющее этот стиль, рисует по внутренней части границы фигуры. Все остальные центрируются по этой границе, так что часть линии оказывается внутри, а часть - снаружи границы.

Ширина (или толщина) пера задается в логических единицах. Это соответствует единицам текущего режима отображения по X-оси. Поскольку мы используем режим отображения MM_TEXT, единицей является пиксель. Задав нулевую толщину пера,

независимо от установленного режима отображения Вы получите перо толщиной ровно в один пиксель.

Большинство устройств GDI не поддерживает широкие линии со стилем - т.е. линии толще 1. Поэтому, запросив точечную линию толщиной в шесть пикселей, Вы, скорее всего, на самом деле получите сплошную линию заданной толщины. Чем "интеллектуальнее" будет становиться аппаратура, тем выше шанс получить линию со стилем неединичной толщины. Пока же для прерывистых линий Вы должны принять это ограничение: толщина у такой линии будет равна одному пикселю.

Как и цвета пикселей, цвета перьев определяются одним из трех способов: RGB-тройкой, индексом палитры или RGB-тройкой относительно палитры. Что бы Вы не предпочли, цвета перьев выбираются в соответствии с чистыми цветами палитры. Исключение составляет стиль PS_INSIDEFRAME, допускающий смешанные цвета.

Смешанные цвета образуются объединением двух или более цветов. В реальном мире в этом нет ничего нового. Оттенок серого часто получают, смешивая черный цвет с белым.

GDI создает смешанные цвета, объединяя два (или более) цвета нормальной палитры для создания на 16-цветном устройстве (например, адаптере EGA или VGA) оттенки сотен разных цветов. На монохромных устройствах типа Hercules или CGA такая методика позволяет создавать десятки оттенков серого. В результате смешения цветов часто бывает трудно без специальных средств, например, без программы Magnify, отличить смешанный цвет от чистого.

Поскольку смешанные цвета несколько усложняют работу драйвера устройства, то их поддерживает всего один стиль PS_INSIDEFRAME. Обычно смешение цветов используется при работе с кистями - их мы будем рассматривать в связи с созданием закрашиваемых фигур.

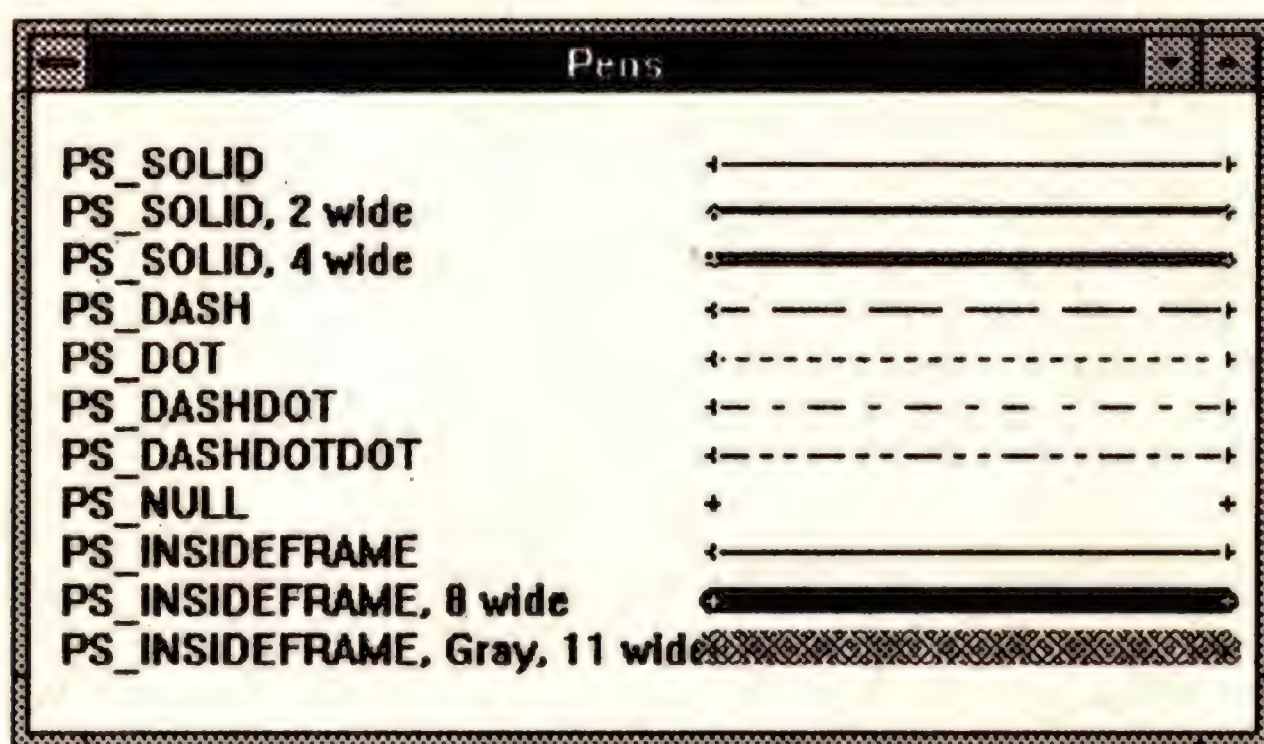


Рис.8.6. Линии, рисуемые с помощью перьев GDI.

Программа Pens рисует десять линий разных стилей, конечные точки которых отмечены маркерами. Отметим для толстых линий, что их маркеры появляются "внутри" линий. Кроме того, отметим, что конец каждой линии скруглен. С помощью программы Magnify Вы можете увидеть, что линия PS_INSIDEFRAME на самом деле нарисована смешанными цветами. Ниже приводится листинг программы Pens:

```
MAKEFILE.MAK
```

```
.AUTODEPEND
```

```
# Определения компилятора
```

```
INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;C:\BORLANDC\INCLUDE
```

```
CC = bcc -c -D_CLASSDLL -H -ml -WS -w -IS(INC)
```

```
# Неявные правила
```

```
.c.obj:
```



```
$(CC) {$< }

.cpp.obj:
$(CC) {$< }

# Явные правила
Pens.exe: Pens.res Pens.def Pens.obj
    tlink /c /C /n /P- /Twe /x @Pens.LNK
    RC Pens.res Pens.exe

# Зависимости
Pens.obj: Pens.cpp
Pens.res: Pens.rc Pens.cur Pens.ico
    RC -R -FO Pens.res Pens.RC
```

PENS.LNK

```
c:\borlandc\lib\c0wl.obj+
Pens.obj
Pens,Pens
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwl.lib+
import.lib+
mathl.lib+
cl.lib
Pens.def
```

PENS.CPP

```
/*
PENS.CPP Прорисовка различных типов перьев GDI
*/
#include <owl.h>

/*
Константы
*/
const int PENCOUNT = 11;
const int MARKERSIZE = 3;

/*
Определения структур данных
*/
typedef struct tagPENDATA {
    WORD        wStyle;
    int         jWidth;
    COLORREF    crColor;
    char        * pchName;
} PENDATA;

PENDATA pens[PENCOUNT] = {
    { PS_SOLID, 1, RGB (0,0,0), "PS_SOLID"},
    { PS_SOLID, 2, RGB (0,0,0), "PS_SOLID, 2 wide"},
    { PS_SOLID, 4, RGB (0,0,0), "PS_SOLID, 4 wide"},
    { PS_DASH, 1, RGB (0,0,0), "PS_DASH"},
    { PS_DOT, 1, RGB (0,0,0), "PS_DOT"},
    { PS_DASHDOT, 1, RGB (0,0,0), "PS_DASHDOT"},
    { PS_DASHDOTDOT, 1, RGB (0,0,0), "PS_DASHDOTDOT"},
    { PS_NULL, 1, RGB (0,0,0), "PS_NULL"},
}
```



```

{ PS_INSIDEFRAME, 1, RGB (0,0,0), "PS_INSIDEFRAME"},
{ PS_INSIDEFRAME, 8, RGB (0,0,0), "PS_INSIDEFRAME, 8 wide"},
{ PS_INSIDEFRAME, 11, RGB (180,180,180),
    "PS_INSIDEFRAME, Gray, 11 wide"}}};
HPEN hpen[PENCOUNT];

/*
Определения классов
*/
class TPensApplication : public TApplication
{
public:
    TPensApplication (LPSTR lpszName, HANDLE hInstance,
        HANDLE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow);
    virtual void InitMainWindow ();
};

class TPensWindow : public TWindow
{
public:
    TPensWindow (PTWindowsObject pwParent, LPSTR lpszTitle,
        PTModule pmModule);
    ~TPensWindow();
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WNDCLASS&);
    virtual void WMPaint(TMessage& Msg) = [WM_PAINT];
};

/*
Прототипы функций
*/
void DrawMarker (HDC hDC, int x, int y);

/*
Главная функция: WinMain.
*/
int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpszCmdLine, int nCmdShow)
{
    TPensApplication Pens ("Pens", hInstance, hPrevInstance,
        lpszCmdLine, nCmdShow);
    Pens.Run();
    return Pens.Status;
}

/*
Компонентная функция класса прикладных программ
*/
TPensApplication::TPensApplication (LPSTR lpszName,
    HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpszCmdLine, int nCmdShow)
    :TApplication (lpszName, hInstance,
        hPrevInstance, lpszCmdLine, nCmdShow)
{
    /* Здесь инициализируется программа. */
}

```



```

/*
Компонентная функция класса прикладных программ
*/
void TPensApplication::InitMainWindow ()
{
    MainWindow = new TPensWindow (NULL, "Pens", NULL);
}

/*
Компонентная функция класса TPensWindow
*/
TPensWindow::TPensWindow (PTWindowsObject pwParent,
    LPSTR lpszTitle, PTModule pmModule)
    :TWindow (pwParent, lpszTitle, pmModule)
{
    int j;
    for(j=0;j<PENCOUNT;j++)
        hpen[j] = CreatePen (pens[j].wStyle, pens[j].jWidth, pens[j].crColor);
}

/*
Компонентная функция класса TPensWindow
*/
TPensWindow::~TPensWindow ()
{
    int j;
    for(j=0;j<PENCOUNT;j++) DeleteObject (hpen[j]);
}

/*
Компонентная функция класса TPensWindow
*/
LPSTR TPensWindow::GetClassName ()
{
    return "Pens:MAIN";
}

/*
Компонентная функция класса TPensWindow
*/
void TPensWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
}

/*
Компонентная функция класса TPensWindow
*/
void TPensWindow::WMPaint(TMessage&)
{
    DWORD dwTextBox;
    int i;
    int xEnd, xStart, xText;
    int yIncr, yLine, yText;
    PAINTSTRUCT ps;

```



```

RECT r;

BeginPaint (HWindow, &ps);
GetClientRect (HWindow, &r);
yIncr = r.bottom / (PENCOUNT*2);
yText = yIncr;
xText = 10;
dwTextBox = GetTextExtent (ps.hdc, pens[10].pchName,
                           strlen(pens[10].pchName));
for (i=0;i<PENCOUNT;i++)
{
    TextOut (ps.hdc, xText, yText, pens[i].pchName, strlen(pens[i].pchName));

    xStart = xText + LOWORD(dwTextBox);
    xEnd   = r.right - 10;
    yLine  = yText + HIWORD(dwTextBox)/2;
    SelectObject (ps.hdc, hpen[i]);
    MoveTo (ps.hdc, xStart, yLine);
    LineTo (ps.hdc, xEnd, yLine);

    /* Прорисовка маркеров. */
    DrawMarker (ps.hdc, xStart, yLine);
    DrawMarker (ps.hdc, xEnd, yLine);
    yText += yIncr;
}

EndPoint (HWindow, &ps);
}

/*
Функция прорисовки маркера
*/
void DrawMarker (HDC hdc, int x, int y)
{
    DWORD dwColor;
    int i;

    dwColor = GetPixel (hdc, x, y);
    dwColor = ~dwColor;
    SetPixel (hdc, x, y, dwColor);
    for (i=0; i< MARKERSIZE; i++)
    {
        dwColor = GetPixel (hdc, x+i, y);
        dwColor = ~dwColor;
        SetPixel (hdc, x+i, y, dwColor);

        dwColor = GetPixel (hdc, x-i, y);
        dwColor = ~dwColor;
        SetPixel (hdc, x-i, y, dwColor);

        dwColor = GetPixel (hdc, x, y+i);
        dwColor = ~dwColor;
        SetPixel (hdc, x, y+i, dwColor);

        dwColor = GetPixel (hdc, x, y-i);
        dwColor = ~dwColor;
        SetPixel (hdc, x, y-i, dwColor);
    }
}

```


PENS.RC

snapshot icon Pens.ico
hand cursor Pens.cur

PENS.DEF

NAME PENS
EXETYPE WINDOWS
DESCRIPTION 'Pens -- Draws with various GDI pens'
CODE MOVEABLE DISCARDABLE
DATA MOVEABLE MULTIPLE
HEAPSIZE 512
STACKSIZE 5120

Данная программа в ответ на сообщение WM_CREATE создает 10 перьев и разрушает их в ответ на сообщение WM_DESTROY. Это существенно. Каждая программа под Windows должна правильно освобождать перья и другие ненужные ей объекты. Проблема возникает в связи с одной особенностью перьев: возможностью их совместного использования.

Чтобы объекты GDI могли использоваться совместно, Windows не отслеживает автоматически, чем они заняты. Поэтому если Вы забыли при завершении программы освободить перья, то неосвобожденная память теряется навсегда (по крайней мере, до выключения питания). Не забывайте освобождать перья, когда они Вашей программе больше не нужны!

Теперь обратим внимание на последний графический атрибут контекста устройства, который играет важную роль при рисовании линий - режим графического вывода. Предварительно остановимся на программе ввода с мыши, которая будет подробно рассмотрена в главе 16.

Режимы графического вывода

Режим графического вывода представляет собой логическую операцию, которая подсказывает GDI, как рисовать пиксели, линии и закрашенные фигуры. Иногда режим графического вывода называют растровой операцией, или сокращенно ROP. Он определяет, как рисуемые пиксели будут взаимодействовать с уже имеющимися.

Для линий режим графического вывода описывает взаимодействие перьев с уже имеющимися на экране пикселями. В реальном мире шариковая ручка проводит линию поверх любых имеющихся на поверхности изображений. Некоторые режимы так и работают, однако набор предоставляемых GDI возможностей намного шире, чем возможности шариковой ручки. В предыдущей главе программа Marker использовала операцию NOT, позволяющую гарантировать видимость маркера на экране. Режим графического вывода обеспечивает быстрое и удобное средство достижения того же эффекта. Высокая скорость вывода достигается за счет того, что логическая операция встроена в драйвер устройства.

Кроме того, режим графического вывода дает возможность рисовать формы, кажущиеся плывущими по экрану. Это, например, относится к курсору мыши, который не должен оставлять за собой следа из "грязных" пикселей. В главе 16 мы рассмотрим программу, которая перемещает объекты по экрану.

Применение алгебры логики в отношении графического вывода может показаться странным. Однако в цифровом компьютере каждый элемент данных кодируется в виде числа. Например, RGB-тройки - это тоже числа, с помощью которых кодируются цветовые составляющие. В графических устройствах, таких как дисплей EGA, числами представлены пиксели, образующие графические образы.

Режимы графического вывода используют это преимущество, позволяя программам выполнять логические операции, которые несложно распространить и на графические образы.



Рис.8.7. Один из режимов: перемещение объектов.

Ниже перечисляются 16 поддерживаемых GDI растровых операций. Отметим, что все они различаются между собой. Если хотите, можете запретить вывод, задав режим R2_NOP, или, задав режим R2_NOT, гарантировать, что рисунок будет сделан. Учтите, что режимы R2_BLACK и R2_WHITE, игнорируют цвет пера.

```
R2_NOP
R2_MASKNOTPEN
R2_MERGE PEN
R2_XORPEN
R2_MASKPEN
R2_BLACK
R2_COPYPEN
R2_MASKPENNOT
R2_MERGE NOTPEN
R2_NOTCOPYPEN
R2_WHITE
R2_NOTMASKPEN
R2_NOTXORPEN
R2_NOTMERGE PEN
R2_MERGE PENNOT
R2_NOT
```

Об атрибутах графического вывода нужно помнить, что они не только влияют на линии, но и играют важную роль при выводе пикселей (подпрограмма SetPixel) и закрашенных геометрических фигур (примитивы, описанные в следующей главе.) Однако при выводе на экран текстов GDI не использует режим графического вывода. Дело здесь в быстродействии. Хотя растровые операции реализованы на уровне драйвера устройства и работают достаточно быстро, они замедляют вывод текстов и потому не используются подпрограммами GDI, выполняющими такой вывод.

Определить текущую установку графического вывода позволяет подпрограмма GetROP2. Для установки в контексте устройства нового значения режима графического вывода Вы вызываете ее:


```
SetROP2 (hdc, R2_XORPEN);
```

Вот более новая и быстросействующая версия подпрограммы рисования маркера, выполненная с использованием ROP:

```
VOID DrawMarker (HDC hdc, int x, int y)
{
    int i;
    int rop;

    rop = SetROP2 (hdc, R2_NOT);
    SetPixel (hdc, x, y, RGB (0, 0, 0));
    for (i=1; i<=MARKERSIZE; i++)
    {
        SetPixel (hdc, x+i, y, RGB (0, 0, 0));
        SetPixel (hdc, x-i, y, RGB (0, 0, 0));
        SetPixel (hdc, x, y+i, RGB (0, 0, 0));
        SetPixel (hdc, x, y-i, RGB (0, 0, 0));
    }
    SetROP2 (hdc, rop);
}
```

Вспомните, что подпрограмма SetPixel выводит изображение цветом, задаваемым в последнем параметре. В данном случае выбран черный цвет RGB(0,0,0). Однако, поскольку мы используем режим графического вывода R2_NOT, GDI игнорирует заданный цвет и инвертирует пиксель, имеющийся на экране.

В следующей главе мы рассмотрим закрашенные фигуры. Вы увидите, что все сказанное относительно линий равным образом относится и к созданию границ закрашиваемых фигур.

Глава 9. Рисование закрашенных фигур

Следующий рассматриваемый нами набор подпрограмм графического вывода создает закрашенные фигуры. Закрашенная фигура состоит из двух частей: собственно закрашенной области и границы вокруг нее. Закрашенные фигуры называют также замкнутыми фигурами, поскольку начальная и конечная точки граничной линии фигуры совпадают.

Примитивы GDI закрашенных фигур могут иметь неодинаковые толщину и стиль граничной линии, что объясняется применением различных перьев. Ведь, вообще говоря, граница фигуры - это просто линия, и для ее рисования GDI использует перо.



Рис.9.1. Примеры закрашенных фигур, выполненных с помощью GDI.

Отметим, что внутренние области фигур тоже допускают большое многообразие. Это результат работы другого объекта GDI - кисти. Подобно тому, как разные перья создают разные линии, разные кисти по-разному закрашивают области фигур.

Чтобы понять, как GDI рисует заполненные фигуры, важно понимать, каким образом интерпретируются координаты. Координаты закрашенных фигур несколько отличаются от используемых при рисовании линий. Если Вы используете оба типа фигур, то должны сделать установки, обеспечивающие выравнивание фигур нужным образом.

В мире графического программирования существует два основных способа интерпретации координат. Несколько в упрощенном виде вопрос ставится так: лежат ли точки пересечения координат в центре пикселей или же пиксели "вписаны" в ячейки координатной сетки.

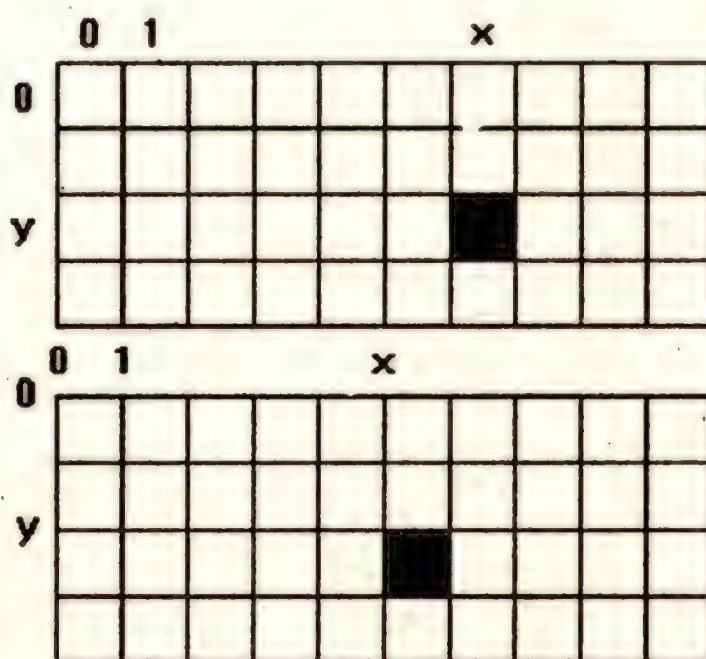


Рис.9.2. Два типа координат GDI: координаты в центре пикселя; координаты в узлах сетки; точки координат; пиксел.

Все примитивы рисования линий GDI используют координаты с точками пересечения в центре пикселей. Из семи примитивов рисования закрашенных фигур GDI два используют координаты с точками пересечения в центре пикселей (пиксельные координаты), а пять - координаты с пикселями, вписанными в ячейки координатной

сетки (сеточные координаты). Каждый из типов координат имеет свои особенности, диктующие его применение в тех или иных случаях. В табл.9.1 показано семь примитивов для закрашивания фигур с используемыми ими типами координат.

Таблица 9.1. Подпрограммы GDI для рисования закрашенных фигур.

Подпрограмма	Координаты	Описание
Polygon	Пиксельные	Закрашенная полилиния
PolyPolygon	То же	Множественные многоугольники
Chord	Сеточные	Частичная дуга, концы которой соединены прямой линией
Ellipse	То же	Полная дуга
Pie	— " —	"Кусок пирога"
Rectangle	— " —	Прямоугольник
RoundRect	— " —	Прямоугольник со скругленными углами

Подпрограммы Polygon и PolyPolygon используют пиксельные координаты. Эти подпрограммы можно считать расширениями Polyline, одного из примитивов GDI для рисования линий. Данные подпрограммы легко использовать совместно с подпрограммами, рисующими линии, так как у них общий тип координат.

Следующие пять подпрограмм используют сеточные координаты, которые служат для определения областей вырезки. Это упрощает использование данных подпрограмм с подпрограммами вырезки GDI.

GDI использует пиксельные координаты, когда программа ориентирована на работу с линиями, и сеточные - для работы с плоскими областями. Однако, если Вы хотите использовать подпрограммы обоих типов, несложно модифицировать координаты одного типа так, чтобы они могли использоваться совместно с координатами другого типа.

Видимый эффект сеточных координат состоит в том, что закрашенные фигуры кажутся на один пиксель меньше, чем Вы ожидали. Вы можете воспользоваться программой Magnify.

Например, рассмотрим графический вывод из программы Rect. Эта программа использует для рисования прямоугольника два сообщения мыши WM_LBUTTONDOWN и WM_LBUTTONUP. Каждое из них определяет по одной из пары точек, необходимых подпрограмме Rectangle. Rectangle принимает пять параметров:

Rectangle (hDC, X1, Y1, X2, Y2);

- * hDC - это логический номер контекста устройства.
- * координаты X1,Y1 определяют один угол прямоугольника.
- * координаты X2,Y2 определяют второй угол прямоугольника.

Рисуемый этой подпрограммой прямоугольник имеет границы, параллельные осям X и Y, поэтому прямоугольник задается всего двумя (а не четырьмя) точками. Два заданных угла лежат напротив друг друга в построенном прямоугольнике.

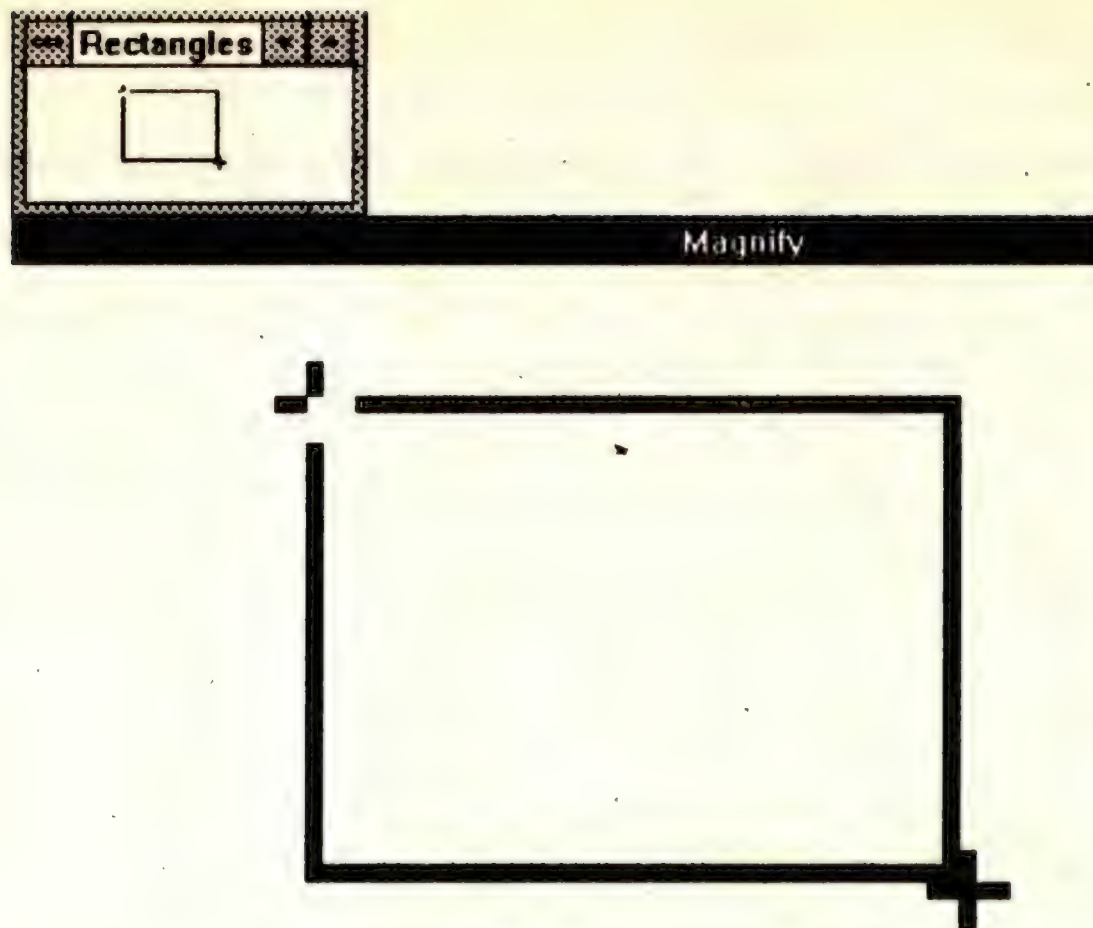


Рис.9.3. Сеточные координаты кажутся меньшими по сравнению с пиксельными на один пиксель.

Вот исходный текст программы, которая рисует нам прямоугольник:

MAKEFILE.MAK

.AUTODEPEND

Определения транслятора

INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;C:\BORLANDC\INCLUDE

CC = bcc -c -D_CLASSDLL -H -ml -WS -w -I\$(INC)

Неявные правила

.c.obj:

\$(CC) {\$< }

.cpp.obj:

\$(CC) {\$< }

Явные правила

Rect.exe: Rect.res Rect.def Rect.obj

tlink /c/C/n/P-/Twe/x @Rect.LNK

RC Rect.res Rect.exe

Зависимости отдельных файлов

Rect.obj: Rect.cpp

Rect.res: Rect.rc Rect.cur Rect.ico

RC -R -FO Rect.res Rect.RC

RECT.LNK

c:\borlandc\lib\c0wl.obj+
 Rect.obj
 Rect,Rect
 \borlandc\owl\lib\owl.lib+
 crtll.lib+
 cwl.lib+
 import.lib+
 mathl.lib+
 cl.lib
 Rect.def

RECT.CPP

/*

RECT.CPP - Рисование набора прямоугольников в ответ на сообщения от мыши

*/

#include <owl.h>

#define MOUSEX(arg) (arg.LP.Lo)

#define MOUSEY(arg) (arg.LP.Hi)

/*

Константы

*/

const int MAXRECTANGLES = 50;

const int MARKERSIZE = 3;

/*

Определения классов

*/

class TRectApplication : public TApplication

{

public:

TRectApplication (LPSTR lpszName, HANDLE hInstance,
HANDLE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow);

virtual void InitMainWindow ();

};

class TRectWindow : public TWindow

{

public:

TRectWindow (PTWindowsObject pwParent, LPSTR lpszTitle,
PTModule pmModule);

virtual LPSTR GetClassName ();

virtual void GetWindowClass (WNDCLASS&);

virtual void Paint(HDC hdc, PAINTSTRUCT& ps);

virtual void WMLButtonDown(TMessage& Msg)=[WM_LBUTTONDOWN];

virtual void WMLButtonUp(TMessage& Msg)=[WM_LBUTTONUP];

private:

RECT arRectangles [MAXRECTANGLES];

int cRects;

};

/*

Прототипы функций

*/

void DrawMarker (HDC hdc, int x, int y);

/*

Основная функция: WinMain

*/

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
LPSTR lpszCmdLine, int nCmdShow)

{

TRectApplication Rect ("Rect", hInstance, hPrevInstance,
lpszCmdLine, nCmdShow);

Rect.Run();

return Rect.Status;

}


```

/*
Компонентная функция класса прикладной программы
*/
TRectApplication::TRectApplication (LPSTR lpszName, HANDLE hInstance,
                                     HANDLE hPrevInstance, LPSTR lpszCmdLine,
                                     int nCmdShow)
    :TApplication (lpszName, hInstance, hPrevInstance,
                  lpszCmdLine, nCmdShow)
{
    /* Здесь инициализируется программа. */
}

/*
Компонентная функция класса прикладной программы
*/
void TRectApplication::InitMainWindow ()
{
    MainWindow = new TRectWindow (NULL, "Rectangles", NULL);
}

/*
Компонентная функция класса TRectWindow
*/
TRectWindow::TRectWindow (PTWindowsObject pwParent,
                          LPSTR lpszTitle, PTModule pmModule)
    :TWindow (pwParent, lpszTitle, pmModule)
{
    cRects = 1;
    arRectangles[0].left=40;
    arRectangles[0].right=80;
    arRectangles[0].top=10;
    arRectangles[0].bottom=40;
}

/*
Компонентная функция класса TRectWindow
*/
LPSTR TRectWindow::GetClassName ()
{
    return "Rect:MAIN";
}

/*
Компонентная функция класса TRectWindow
*/
void TRectWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
}

/*
Компонентная функция класса TRectWindow
*/
void TRectWindow::WMLButtonDown(TMessage& Msg)
{
    arRectangles[cRects].left = MOUSEX(Msg);
}

```



```

    arRectangles[cRects].top = MOUSEY(Msg);
}

/*
Компонентная функция класса TRectWindow
*/
void TRectWindow::WMLButtonUp(TMessage& Msg)
{
    arRectangles[cRects].right = MOUSEX(Msg);
    arRectangles[cRects].bottom = MOUSEY(Msg);
    cRects++;
    if (cRects == MAXRECTANGLES) cRects = 0;
    InvalidateRect (HWindow, NULL, TRUE);
}

/*
Компонентная функция класса TRectWindow
*/
void TRectWindow::Paint (HDC hdc, PAINTSTRUCT& ps)
{
    int i;
    for (i = 0; i < cRects; i++)
    {
        Rectangle (hdc, arRectangles[i].left,
                    arRectangles[i].top,
                    arRectangles[i].right,
                    arRectangles[i].bottom);
        DrawMarker (hdc, arRectangles[i].left,
                    arRectangles[i].top);
        DrawMarker (hdc, arRectangles[i].right,
                    arRectangles[i].bottom);
    }
}

/*
Функция прорисовки маркера
*/
void DrawMarker (HDC hdc, int x, int y)
{
    int i;
    int rop;

    rop = SetROP2 (hdc, R2_NOT);
    SetPixel (hdc, x, y, RGB(0, 0, 0));
    for (i=1; i < MARKERSIZE; i++)
    {
        SetPixel (hdc, x+i, y, RGB(0, 0, 0));
        SetPixel (hdc, x-i, y, RGB(0, 0, 0));
        SetPixel (hdc, x, y+i, RGB(0, 0, 0));
        SetPixel (hdc, x, y-i, RGB(0, 0, 0));
    }
    SetROP2 (hdc, rop);
}

```

RECT.RC

snapshot icon Rect.ico
hand cursor Rect.cur

RECT.DEF

NAME RECT

EXETYPE WINDOWS

DESCRIPTION 'Rect -- Rectangle Drawing'

CODE MOVEABLE DISCARDABLE

DATA MOVEABLE MULTIPLE

HEAPSIZE 512

STACKSIZE 5120

Для хранения точек, описывающих каждый прямоугольник, мы выделили массив типа RECT:

```
RECT arRectangles[MAXRECTANGLES];
```

RECT определен в WINDOWS.H как:

```
typedef struct tagRECT
```

```
{
```

```
    int left;
```

```
    int top;
```

```
    int right;
```

```
    int bottom;
```

```
} RECT;
```

Названия элементов массива могут показаться несколько необычными, так как они отражают использование структуры для прямоугольников вырезки. Поля left и right содержат значения X, а поля top и bottom - Y.

Мы используем макропроцедуру MAKEPOINT для копирования позиции мыши из параметра lParam сообщения мыши в переменную POINT. Когда пользователь нажимает левую кнопку мыши, генерируется сообщение WM_LBUTTONDOWN. После того как наш оконный объект принял это сообщение, он помещает координаты мыши в поля left и top в arRectangles. Когда кнопка мыши будет отпущена, генерируется сообщение WM_LBUTTONUP. Оконный объект реагирует на него помещением информации о позиции мыши в поля right и bottom в arRectangles.

В ответ на сообщение WM_LBUTTONUP оконный объект с помощью подпрограммы InvalidateRect объявляет все окно недействительным. Вспомните, это приводит к генерации сообщения WM_PAINT, что позволяет держать вместе все строки кода, относящиеся к рисованию прямоугольника.

В ответ на сообщение WM_PAINT компонентная функция TWindow WMPaint выполняет всю стандартную работу для этого сообщения. Затем она вызывает компонентную функцию Paint, которую оконный объект переопределяет для использования с окном. Минимум, что требуется для обработки WM_PAINT, - вызвать BeginPaint и EndPaint. Между этими вызовами в окне можно рисовать что угодно.

Наша компонентная функция Paint пробегает по массиву прямоугольников, поочередно рисуя каждый из них. После того как прямоугольник нарисован, в двух определяющих его точках помещаются маркеры.

Это краткое введение в подпрограмму Rectangle, а также знакомство с двумя разными типами координат GDI подготовили нас к изучению остальных подпрограмм GDI, предназначенных для рисования закрашенных фигур.

Подпрограммы GDI для рисования закрашенных фигур

Рассмотрим поближе каждую из подпрограмм GDI для рисования закрашенных фигур. Их всего семь: Polygon, PolyPolygon, Chord, Ellipse, Pie, Rectangle и RoundRect.

Начнем с двух подпрограмм, работающих в пиксельных координатах.

Подпрограммы Polygon и PolyPolygon

Подпрограмма Polygon определена как:

Polygon (hDC, lpPoints, nCount)

- * hDC - логический номер контекста устройства;
- * lpPoints - указатель на массив типа POINT. Он описывает соединяемые точки;
- * nCount - это число соединяемых точек.

Подобно подпрограмме Polyline, о которой шла речь в главе 8, подпрограмма Polygon соединяет расположенные последовательно точки с помощью текущего установленного в контексте устройства пера. Если первая и последняя точки не совпадают, то проводится линия, замыкающая фигуру, а область внутри нее закрашивается текущей установленной в контексте устройства кистью.

Подпрограмма Polygon - это одна из наиболее гибких подпрограмм, рисующих закрашенные фигуры, так как с ее помощью Вы можете нарисовать практически любую фигуру. В отличие от обычной подпрограммы Rectangle, подпрограмма Polygon может рисовать повернутые многоугольники (рис.9.4).

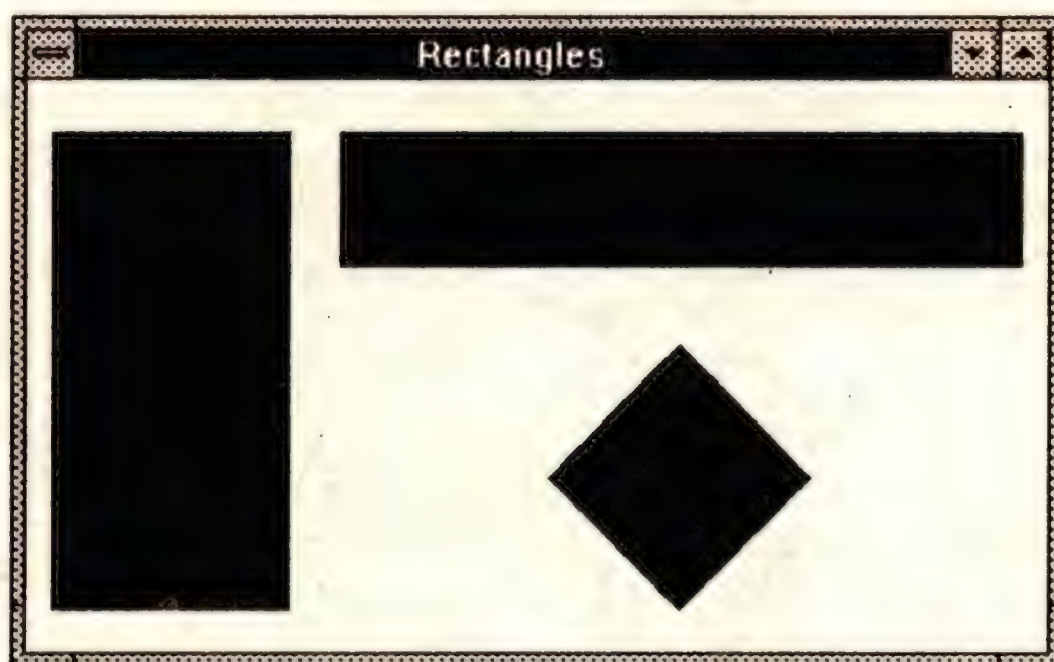


Рис.9.4. Прямоугольники, нарисованные с помощью подпрограммы Polygon.

Ниже приводится фрагмент кода для рисования многоугольников:

```
void TSapleWindow::Paint (HDC hdc, PAINTSTRUCT& ps)
{
    SelectObject (hdc, GetStockObject (BLACK_BRUSH));

    apt[0].x = 10;   apt[0].y = 20;
    apt[1].x = 100;  apt[1].y = 20;
    apt[2].x = 100;  apt[2].y = 200;
    apt[3].x = 10;   apt[3].y = 200;
    Polygon (hdc, apt, 4);

    apt[0].x = 120;  apt[0].y = 20;
    apt[1].x = 380;  apt[1].y = 20;
    apt[2].x = 380;  apt[2].y = 70;
    apt[3].x = 120;  apt[3].y = 70;
    Polygon (hdc, apt, 4);

    apt[0].x = 200;  apt[0].y = 150;
    apt[1].x = 250;  apt[1].y = 100;
    apt[2].x = 300;  apt[2].y = 150;
    apt[3].x = 250;  apt[3].y = 200;
    Polygon (hdc, apt, 4);
}
```


Подпрограмма PolyPolygon позволяет нарисовать за один вызов несколько многоугольников. Она является расширением подпрограммы Polygon. Подпрограмма PolyPolygon, рисующая сразу несколько многоугольников, работает быстрее, чем Polygon, которая для этой цели должна сделать несколько вызовов эквивалентных функций, отсюда возникает разница в скорости.

Подпрограмма PolyPolygon определена как:

PolyPolygon (hDC, lpPoints, lpPolyCount, nCount)

- * hDC - логический номер контекста устройства;
- * lpPoints - указатель на массив типа POINT. Он описывает соединяемые точки всех создаваемых многоугольников;
- * lpPolyCount - указатель на массив типа INT. Элементы массива задают число точек в каждом многоугольнике;
- * nCount - целое число, задающее количество точек в массиве lpPolyCount. Другими словами, это число рисуемых многоугольников.

Ниже приведен вариант предыдущего примера для PolyPolygon:

```
void TSapleWindow::Paint (HDC hdc, PAINTSTRUCT& ps)
{
    SelectObject (hdc, GetStockObject (BLACK_BRUSH));
    /* Первый многоугольник */
    apt[0].x = 10;   apt[0].y = 20;
    apt[1].x = 100;  apt[1].y = 20;
    apt[2].x = 100;  apt[2].y = 200;
    apt[3].x = 10;   apt[3].y = 200;
    apt[4].x = 10;   apt[4].y = 20;
    ai[0] = 5;
    /* Второй многоугольник */
    apt[5].x = 120;  apt[5].y = 20;
    apt[6].x = 380;  apt[6].y = 20;
    apt[7].x = 380;  apt[7].y = 70;
    apt[8].x = 120;  apt[8].y = 70;
    apt[9].x = 120;  apt[9].y = 20;
    ai[1] = 5;
    /* Третий многоугольник */
    apt[10].x = 200; apt[10].y = 150;
    apt[11].x = 250; apt[11].y = 100;
    apt[12].x = 300; apt[12].y = 150;
    apt[13].x = 250; apt[13].y = 200;
    apt[14].x = 200; apt[14].y = 150;
    ai[2] = 5;
    PolyPolygon (hdc, apt, ai, 3);
}
```

Говоря о многоугольниках, мы обычно имеем в виду фигуры с равными сторонами, например создаваемые показанными фрагментами прямоугольники. Однако, используя подпрограмму Polygon, можно рисовать и кривые при условии, что мы зададим достаточное число точек. GDI именно таким образом, при помощи множества коротких линий, моделирует кривые. Разумеется, если бы мы решили сами создавать кривые подобным способом, то вынуждены были бы сами рассчитывать каждую ее точку.

Однако GDI имеет набор подпрограмм, облегчающих эту процедуру, рисует закрашенные фигуры, контурами которых являются кривые.

Подпрограммы Ellipse, Chord, Pie

GDI имеет три подпрограммы для создания закрашенных фигур, контурами которых являются кривые. Их можно рассматривать как расширение подпрограммы Arc из предыдущей главы. Каждая подпрограмма использует ограничивающий прямоугольник, в который должна вписываться фигура.

Ниже показано определение функции Ellipse:

Ellipse (hDC, X1, Y1, X2, Y2)

- * hDC - логический номер контекста устройства;
- * (X1,Y1) - координаты одного угла ограничивающего прямоугольника;
- * (X2,Y2) - координаты противоположного угла ограничивающего прямоугольника.

Функция Ellipse рисует эллипс, который вписан в ограничивающий прямоугольник.

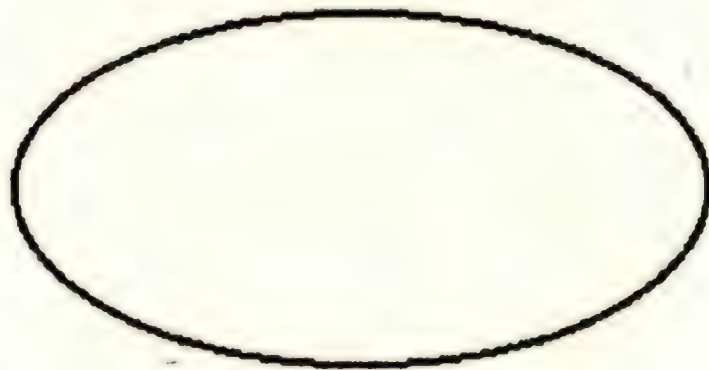


Рис.9.5. Фигура, нарисованная с помощью подпрограммы Ellipse.

Функция Chord также использует ограничивающий прямоугольник для рисования частичной дуги, концы которой соединены сегментом линии (рис.9.6). Определение функции Chord следующее:

Chord (hDC, X1, Y1, X2, Y2, X3, Y3, X4, Y4)

- * hDC - логический номер контекста устройства;
- * (X1,Y1) и (X2,Y2) - координаты ограничивающего прямоугольника;
- * (X3,Y3) - начальная точка сегмента линии;
- * (X4,Y4) - конечная точка сегмента линии.



Рис.9.6. Фигура, нарисованная с помощью подпрограммы Chord.

Функция Pie принимает те же параметры, что и Chord, но рисует не хорду эллипса, а "кусочек пирога", вырезанный из эллипса, т.е. его сектор (рис.9.7). Она определена следующим образом:

Pie (hDC, X1, Y1, X2, Y2, X3, Y3, X4, Y4)

- * hDC - логический номер контекста устройства;
- * (X1,Y1) и (X2,Y2) - координаты ограничивающего прямоугольника;
- * (X3,Y3) - начальная точка сектора;
- * (X4,Y4) - конечная точка сектора.



Рис.9.7. Фигура, рисуемая с помощью подпрограммы Pie

Подпрограммы Rectangle и RoundRect

Выше мы рассматривали функцию Rectangle, но для полноты описания приведем ее определение и здесь:

Rectangle (hDC, X1, Y1, X2, Y2)

- * hDC - логический номер контекста устройства;
- * (X1,Y1) - координаты одного угла прямоугольника;
- * (X2,Y2) - координаты противоположного угла прямоугольника.

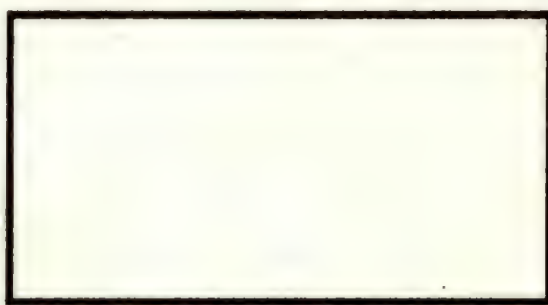


Рис.9.8. Фигура, нарисованная с помощью подпрограммы Rectangle.

Как и прочие подпрограммы, рисующие закрашенные фигуры, Rectangle использует перо из контекста устройства для рисования контура прямоугольника и кисть для закрашивания внутренней его части. Эта подпрограмма рисует прямоугольники только со сторонами, параллельными осям X и Y (рис.9.8). Для повернутых прямоугольников Вы должны пользоваться функцией Polygon или PolyPolygon.

Подпрограмма RoundRect рисует прямоугольник со скругленными углами (рис.9.9). Она определена следующим образом:

RoundRect (hDC, X1, Y1, X2, Y2, X3, Y3)

- * hDC - логический номер контекста устройства;
- * (X1,Y1) - координаты одного угла прямоугольника;
- * (X2,Y2) - координаты противоположного угла прямоугольника;
- * (X3,Y3) - координаты, ограничивающего прямоугольника для эллипса, используемого при скруглении угла.



Рис.9.9. Фигура, нарисованная с помощью подпрограммы RoundRect.

Атрибуты контекста устройства

Для того чтобы понять, насколько полный контроль дает нам над создаваемыми закрашенными фигурами GDI, требуется изучение контекста устройства. Ниже приводится список атрибутов, которые влияют на вывод закрашенных фигур:

Атрибут графического вывода	Комментарии
Цвет фона	Второй цвет для штрихующих кистей и прерывистых перьев
Режим фона	Включает/выключает отображение цвета фона
Кисть	Цвет заполнения внутренней области фигуры
Точка привязки кисти	Выравнивание штрихующей кисти
Режим графического вывода	Логическая операция при рисовании
Перо	Цвет, ширина и стиль контура фигуры
Режим закрашивания многоугольника	Для подпрограмм Polygon и PolyPolygon

Поскольку границами (контурами) закрашенных фигур являются линии, все атрибуты контекста устройства, применимые к линиям, применимы и к контурам фигур: цвет фона, режим отображения фона, режим графического вывода и перо. Важно, что при использовании для рисования контура перо всегда центрируется по этому контуру. Например, если перо имеет ширину в девять пикселей, то четыре из них окажутся вне фигуры, четыре внутри фигуры, а один на самом контуре.

Однако перья, создаваемые со стилем PS_INSIDEFRAME, никогда не выходят за границу фигуры. Например, девятипиксельное перо со стилем PS_INSIDEFRAME нарисует линию, у которой один пиксель будет идти строго по контуру фигуры, а остальные восемь будут лежать внутри нее. Это важно для создания фигур, которые должны занимать определенную область, не выходя за нее.

Следующие атрибуты являются специфичными для закрашенных фигур: кисть, точка привязки кисти, а также режим закрашивания многоугольника. Прежде чем рассматривать кисти, мы исследуем два атрибута: режим закрашивания многоугольника и режим графического вывода.

Имена режимов графического вывода выглядят таким образом, как будто они влияют только на перья. Например, режим по умолчанию R2_COPYPEN включает в себя слово "pen" ("перо"), являющееся атрибутом рисования линий. Несмотря на этот не очень удачный выбор имен, режимы графического вывода влияют и на закрашенные фигуры, в том числе на их контур и на закрашенную область. Режим графического вывода равным образом оказывает влияние на перья и кисти.

Режим закрашивания многоугольника определяет, каким образом будет выполнено закрашивание сложных фигур, создаваемых подпрограммами, Polygon и PolyPolygon. Для простых фигур, таких как квадраты и прямоугольники, этот режим никакой роли не играет. Однако для сложных фигур, например звезды, как бы составленной из смежных многоугольников со своими границами, этот режим определяет, какие области будут закрашены.



Рис.9.10. Два режима заполнения многоугольников.

Режим WINDING (сквозное закрашивание) заполняет все фигуры внутри общей границы. Напротив, режим ALTERNATE (чередование) заполняет только нечетные области, т.е. закрашивание производится на пересечении нечетных номеров границ (1, 3, 5) и производится на пересечении четных - (2, 4, 6). GDI не имеет средств для включения закрашивания только четных номеров, но Вы можете добиться этого эффекта, просто нарисовав одну и ту же фигуру дважды (рис.9.10).

Подпрограмма SetPolyFillMode, управляющая этим атрибутом, определена следующим образом:

SetPolyFillMode (hDC, nPolyFillMode)

- * hDC - логический номер контекста устройства;
- * nPolyFillMode - либо ALTERNATE, либо WINDING.

Теперь рассмотрим самый важный с точки зрения внешнего вида закрашенных фигур атрибут: кисть GDI.

О кистях

Кисть представляет собой атрибут графического вывода контекста устройства, определяемый тремя характеристиками: стилем, цветом и шаблоном. Размер кисти составляет 8x8 пикселей. Рассматривая перья GDI, мы упоминали, что групповые атрибуты позволяют одновременно устанавливать несколько атрибутов. Фактор удобства, безусловно, один из основных для введения понятия кисти.

Выше мы отмечали, что перья могут совместно использоваться программами и устройствами. То же самое справедливо и для кистей. Как существуют стандартные перья, так существуют и стандартные кисти, т.е. GDI создает набор кистей, доступных для использования любой программой. Если стандартные кисти не обеспечивают Ваших потребностей, то Вы можете легко создать собственные. И наконец, как и в случае перьев, закончив работу, Вы должны не забывать "помыть кисти", иначе говоря удалить их из памяти. В противном случае эта память будет утеряна для системы. Рассмотрим некоторые детали, связанные с созданием и использованием кистей.

Создание и использование кистей

В момент начальной загрузки системы GDI создает такие стандартные кисти: черную, темно-серую, серую, светло-серую, белую и нулевую (или пустую). Как и нулевое перо, нулевая кисть просто резервирует место. Вы можете считать ее прозрачной.

Подпрограмма GetStockObject позволяет получать логические номера стандартных кистей. Для использования кисти ее нужно вставить в контексте устройства с помощью подпрограммы SelectObject. Вот как можно нарисовать серый внутри прямоугольник:

```
brush GetStockObject (GRAY_BRUSH);  
SelectObject (hdc, brush);  
Rectangle (hdc, X1, Y1, X2, Y2);
```

Стандартные кисти Вы можете дополнять своими собственными. Существует три типа кистей пользователя: сплошные, штриховые и шаблонные. GDI обеспечивает пять подпрограмм, с помощью которых можно создавать кисти: CreateBrushIndirect, CreateDIBPatternBrush, CreateHatchBrush, CreatePatternBrush и CreateSolidBrush.

Рассмотрим сначала первые две подпрограммы из этого списка. Уникальность первой из них, CreateBrushIndirect, состоит в том, что она принимает в качестве параметра указатель на структуру данных, способную описывать любую кисть: LOGBRUSH (логическая кисть). Вторая, CreateDIBPatternBrush, создает кисть из аппаратно-независимой битовой матрицы, называемой DIB. Матрица DIB - это битовая матрица, информация о цветах которой хранится в стандартном формате. Цвета в DIB могут правильно интерпретировать любые устройства GDI.

Обратим внимание на три типа битовых матриц: сплошные, штриховые и шаблонные. Рассматривая каждый из них, мы изучим подпрограммы CreateSolidBrush, CreateHatchBrush и CreatePatternBrush, используемые нашими программами для создания соответствующих типов кисти. Сплошная кисть - это кисть, создаваемая либо из чистого цвета, либо из смешанных цветов, о которых мы говорили в предыдущей главе, когда речь шла о перьях. Смешанные цвета создаются путем смешивания чистых цветов. Предположим, устройство имеет два цвета: черный и белый. При смешивании мы получим множество оттенков серого.

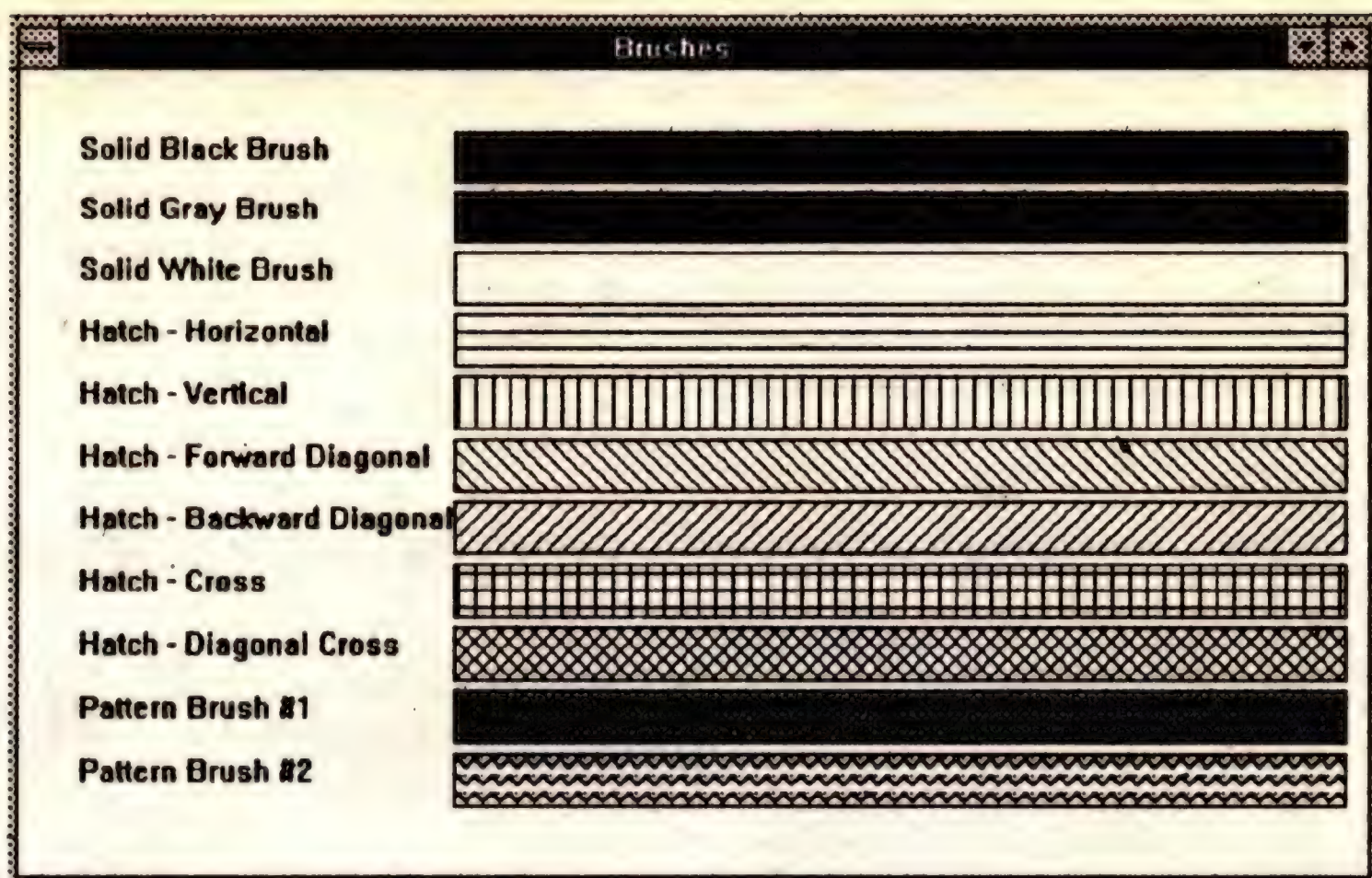


Рис.9.11. Одиннадцать различных кистей.

Подпрограмма `CreateSolidBrush` позволяет создавать такие кисти. Она определена как:

`HBRUSH CreateSolidBrush (crColor)`

* `crColor` обозначает цвет. Он может быть задан RGB-триплетом, индексом в палитре или тройкой RGB относительно палитры.

Например, можно создать белую кисть следующим образом:

`hbr = CreateSolidBrush (GRB (255, 255, 255));`

Штриховая кисть закрашивает области с помощью шаблона, создаваемого из штрихов. GDI имеет шесть встроенных шаблонов штриховки. Подпрограмма `CreateHatchBrush` дает самый простой способ создания штриховой кисти (хотя `CreateBrushIndirect` тоже позволяет выполнить эту операцию).

Синтаксис `CreateHatchBrush` следующий:

`HBRUSH CreateSolidBrush (nIndex, crColor)`

* `nIndex` может принимать одно из шести значений:

`HS_BDIAGONAL`
`HS_CROSS`
`HS_DIAGCROSS`
`HS_FDIAGONAL`
`HS_HORIZONTAL`
`HS_VERTICAL`

* `crColor` обозначает цвет. Он может быть задан тройкой RGB, индексом в палитре или тройкой RGB относительно палитры.

Шаблонная кисть создается на базе битового шаблона. Если шести стилей штриховой кисти для Вас недостаточно, то Вы можете создать шаблонную кисть с любым шаблоном штриховки, которая по внешнему виду может напоминать сплошную кисть, хотя работает совсем по-другому.

Для создания шаблонной кисти служит подпрограмма `CreatePatternBrush`. Ее синтаксис следующий:

`HBRUSH CreatePatternBrush (hBitMap)`

* `hBitMap` - логический номер битовой матрицы.

Для создания шаблонной кисти Вам прежде всего нужна битовая матрица. Существует много способов создания битовых матриц. Сейчас мы ограничимся двумя из них, используемыми в примере программы.

Первый метод создания битовых матриц состоит в использовании графического редактора, такого как Resource Workshop, который позволяет нарисовать шаблон битовой матрицы. Этот шаблон записывается в файл (в нашем примере программы - это SQUARE.BMP). Чтобы включить эту битовую матрицу в программу, в файле сценария ресурсов мы вставляем строку:

```
square bitmap square.bmp
```

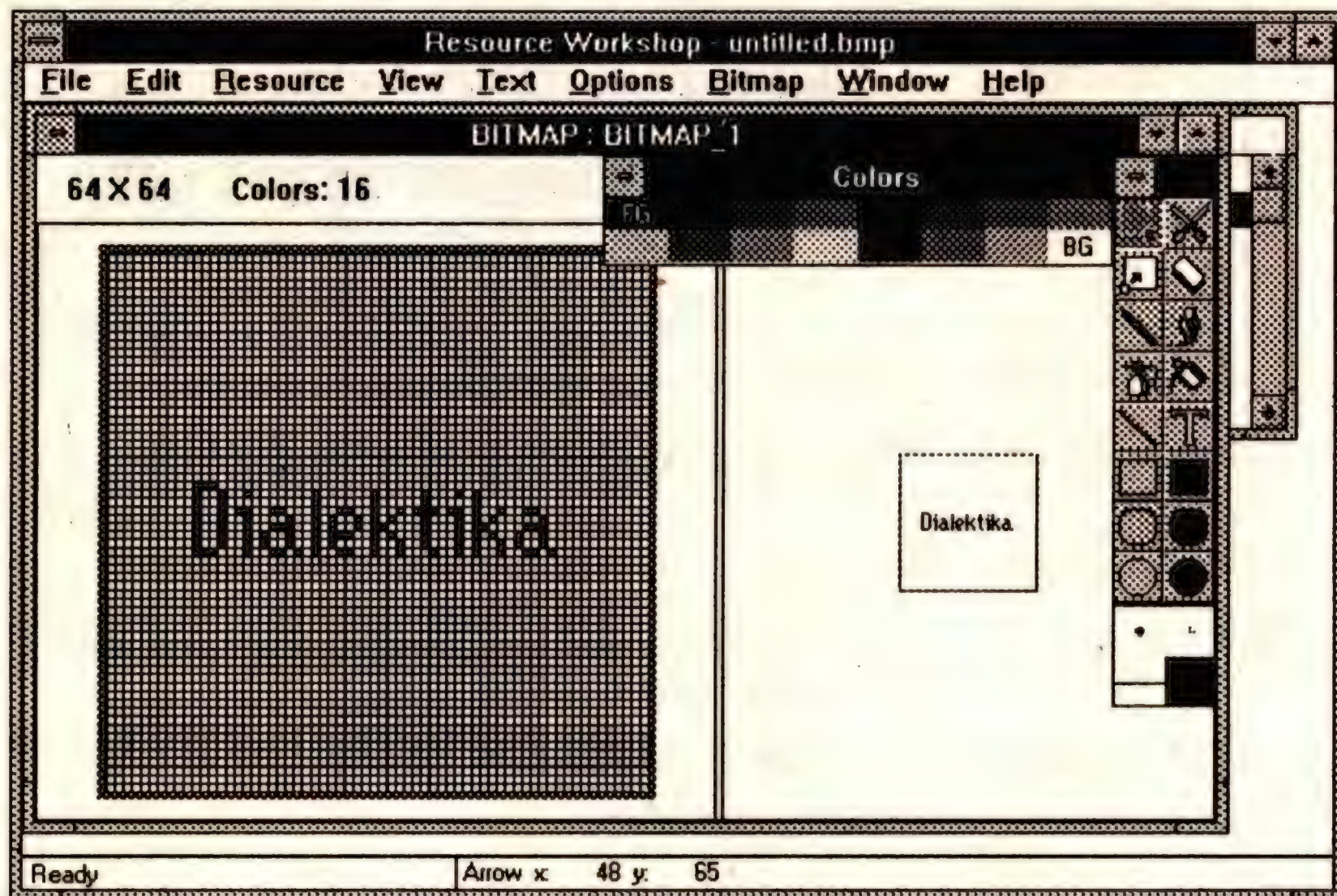


Рис.9.12. Использование Resource Workshop для создания битовой матрицы.

Следующие строки считывают эту битовую матрицу в память и используют ее для создания шаблонной кисти:

```
hbm = LoadBitmap (hInst, "square"); hbr[9] = CreatePatternBrush (hbm);
```

Эта кисть теперь готова для использования при закрашивании внутренних областей фигур GDI.

Второй метод создания битовых матриц состоит в вызове CreateBitmap с указателем на биты, которые должны быть включены в матрицу. Подпрограмма CreateBitmap определена следующим образом:

```
HBITMAP CreateBitmap (nWidth, nHeight, nPlanes, nBitCount, lpBits)
```

* nWidth и nHeight - ширина и высота битовой матрицы в пикселях. Поскольку все кисти GDI имеют размер 8x8 пикселей, то оба поля должны быть равны 8;

* nPlanes - число слоев битовой матрицы. Это один из способов хранения информации о цветах (например, в дисплейном адаптере EGA). Мы собираемся создать монохромную битовую матрицу, поэтому примем это значение nPlanes равным 1;

* nBitCount - число битов, отведенных под цвет. Упакованные пиксели представляют второй способ хранения информации о цветах. Этот метод используется в некоторых режимах дисплейного адаптера CGA. Мы собираемся создать монохромную битовую матрицу, поэтому примем значение nBitCount равным 1;

* lpBits - указатель на биты, используемые для инициализации битовой матрицы.

Наша программа создает битовую матрицу второго шаблона:


```
hbm = CreateBitmap (8, 8, 1, 1, acPattern); hbr[10] = CreatePatternBrush (hbm);
```

Последний параметр CreateBitmap является указателем на символьный массив, определяемый нашей программой как:

```
static unsigned char acPattern[]
(0xFF, 0, /* 1 1 1 1 1 1 1 1 */
0xE7, 0, /* 1 1 1 0 0 1 1 1 */
0xC3, 0, /* 1 1 0 0 0 0 1 1 */
0x99, 0, /* 1 0 0 1 1 0 0 1 */
0x3C, 0, /* 1 1 1 1 1 1 1 1 */
0xFF, 0, /* 1 1 1 1 1 1 1 1 */
0xFF, 0, /* 1 1 1 1 1 1 1 1 */
0xFF, 0); /* 1 1 1 1 1 1 1 1 */
```

Массив определен в шестнадцатичном формате, но для Вашего удобства в комментариях даны двоичные эквиваленты. Значения 1 представляют белые пиксели, а 0 - черные. Поскольку ожидается выравнивание данных битовых матриц по 16-битовой границе, между каждым из значений битовой матрицы появляется дополнительный нулевой байт.

Каждая из этих подпрограмм создает логическую кисть. Подобно логическим перьям, логические кисти - это запросы, которые могут быть переданы любому устройству. При запросе на использование кисти драйвер устройства реализует эту кисть, т.е. преобразует логический запрос в форму, соответствующую возможностям конкретного устройства.

Ниже приведены исходные файлы программы BRUSHES для создания кистей:

MAKEFILE.MAK

.AUTODEPEND

```
# Определения транслятора
INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;
C:\BORLANDC\INCLUDE
CC = bcc -c -D_CLASSDLL -H -ml -WS -w-par -I$(INC)
```

Неявные правила

```
.c.obj:
$(CC) {S< }
.cpp.obj:
$(CC) {S< }
```

Явные правила

```
Brushes.exe: Brushes.res Brushes.def Brushes.obj
tlink /c/C/n/P-/Twe/x @Brushes.LNK
RC Brushes.res Brushes.exe
```

Зависимости отдельных файлов

```
Brushes.obj: Brushes.cpp

Brushes.res: Brushes.rc Brushes.cur Brushes.ico
RC -R -FO Brushes.res Brushes.RC
```

BRUSHES.LNK

```
c:\borlandc\lib\cOwl.obj+
Brushes.obj
Brushes, Brushes
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwl.lib+
```



```
import.lib+
mathl.lib+
cl.lib
Brushes.def
```

BRUSHES.CPP

```
/*
BRUSHES.CPP - Демонстрация различных типов кистей GDI
*/
#include <owl.h>
#include <string.h>

/*
Константы
*/
const int BRUSHCOUNT = 11;
const int PATTERNSIZE = 16;

/*
Объявления классов
*/
class TBrushesApplication : public TApplication
{
public:
    TBrushesApplication (LPSTR lpszName, HANDLE hInstance,
                        HANDLE hPrevInstance,
                        LPSTR lpszCmdLine, int nCmdShow);
    virtual void InitMainWindow ();
};

class TBrushesWindow : public TWindow
{
public:
    TBrushesWindow (PTWindowsObject pwParent,
                    LPSTR lpszTitle, PTModule pmModule);
    ~TBrushesWindow();
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WNDCLASS&);
    virtual void Paint (HDC hdc, PAINTSTRUCT& ps);
private:
    HBRUSH hbr[BRUSHCOUNT];
    char * apszDesc[BRUSHCOUNT];
    unsigned char acPattern[PATTERNSIZE];
};

/*
Основная функция: WinMain
*/
int PASCAL WinMain (HANDLE hInstance,
                    HANDLE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
{
    TBrushesApplication Brushes ("Brushes", hInstance, PrevInstance,
                                lpszCmdLine, nCmdShow);

    Brushes.Run();
    return Brushes.Status;
}
```



```

/*
Компонентная функция класса прикладной программы
*/
TBrushesApplication::TBrushesApplication (LPSTR lpszName,
      HANDLE hInstance,
      HANDLE hPrevInstance,
      LPSTR lpszCmdLine, int nCmdShow)
      :TApplication (lpszName, hInstance, hPrevInstance, lpszCmdLine,
      nCmdShow)
{
    /* Здесь инициализируется прикладная программа */
}

/*
Компонентная функция класса прикладной программы
*/
void TBrushesApplication::InitMainWindow ()
{
    MainWindow = new TBrushesWindow(NULL, "Brushes", NULL);
}

/*
Компонентная функция класса TBrushesWind
*/
TBrushesWindow::TBrushesWindow (PTWindowsObject pwParent,
      LPSTR lpszTitle, PTModule pmModule)
      :TWindow (pwParent, lpszTitle, pmModule)
{
    HBITMAP hbm;

    /*
    Инициализация массива для динамического создания битовой матрицы
    */

    acPattern[0] = 0xff;
    acPattern[1] = 0;
    acPattern[2] = 0xe2;      /* 1 1 1 1 1 1 1 1 */
    acPattern[3] = 0;        /* 1 1 1 0 0 1 1 1 */
    acPattern[4] = 0xc3;     /* 1 1 0 0 0 0 1 1 */
    acPattern[5] = 0;        /* 1 0 0 1 1 0 0 1 */
    acPattern[6] = 0x99;     /* 0 0 1 1 1 1 0 0 */
    acPattern[7] = 0;        /* 0 1 1 1 1 1 1 0 */
    acPattern[8] = 0x3c;     /* 1 1 1 1 1 1 1 1 */
    acPattern[9] = 0;        /* 1 1 1 1 1 1 1 1 */
    acPattern[10] = 0x7e;
    acPattern[11] = 0;
    acPattern[12] = 0xff;
    acPattern[13] = 0;
    acPattern[14] = 0xff;
    acPattern[15] = 0;

    /*
    Создание кистей
    */
    apszDesc[0] = "Solid Black Brush";
    hbr[0] = CreateSolidBrush (RGB (0, 0, 0));
    apszDesc[1] = "Solid Gray Brush";

```



```

hbr[1]          = CreateSolidBrush (RGB (64, 64, 64));
apszDesc[2]     = "Solid White Brush";
hbr[2]          = CreateSolidBrush (RGB (255, 255, 255));

apszDesc[3]     = "Hatch - Horizontal";
hbr[3]          = CreateHatchBrush (HS_HORIZONTAL, RGB(0, 0, 0));

apszDesc[4]     = "Hatch - Vertical";
hbr[4]          = CreateHatchBrush (HS_VERTICAL RGB(0, 0, 0));

apszDesc[5]     = "Hatch - Forward Diagonal";
hbr[5]          = CreateHatchBrush (HS_FDIAGONAL, RGB(0, 0, 0));

apszDesc[6]     = "Hatch - Backward Diagonal";
hbr[6]          = CreateHatchBrush (HS_BDIAGONAL, RGB(0, 0, 0));

apszDesc[7]     = "Hatch - Cross";
hbr[7]          = CreateHatchBrush (HS_CROSS, RGB(0, 0, 0));

apszDesc[8]     = "Hatch - Diagonal Cross";
hbr[8]          = CreateHatchBrush (HS_DIAGCROSS, RGB(0, 0, 0));

apszDesc[9]     = "Pattern Brush #1";
hbm             = LoadBitmap (GetApplication()->hInstance, "square");
hbr[9]          = CreatePatternBrush (hbm);
DeleteObject (hbm);

apszDesc[10]    = "Pattern Brush #2";
hbm             = CreateBitmap (8, 8, 1, 1, (LPSTR)&acPattern[0]);
hbr[10]         = CreatePatternBrush (hbm);
DeleteObject (hbm);
}

/*
Компонентная функция класса TBrushesWind
*/
TBrushesWindow::~TBrushesWindow()
{
    int i;
    for (i=0;i<BRUSHCOUNT;i++) DeleteObject (hbr[i]);
}

/*
Компонентная функция класса TBrushesWind
*/
LPSTR TBrushesWindow::GetClassName ()
{
    return "Brushes:MAIN";
}

/*
Компонентная функция класса TBrushesWind
*/
void TBrushesWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
}

```



```

/*
Компонентная функция класса TBrushesWind
*/
void TBrushesWindow::Paint (HDC hdc, PAINTSTRUCT& ps)
{
    DWORD dw;
    int i;
    int xStart, xEnd, xText;
    int yIncr, yLine, yText;
    int cxWidth;
    RECT r;
    TEXTMETRIC tm;

    GetClientRect (HWindow, &r);
    yIncr = r.bottom / (BRUSHCOUNT+2);
    yText = yIncr;

    /* Определяет размер четырех пробелов для выравнивания текста */
    GetTextMetrics (hdc, &tm);
    xText = tm.tmAveCharWidth * 4;

    /* Определяет размер максимальной строки */
    dw = GetTextExtent (hdc, apszDesc[6], strlen(apszDesc[6]));
    cxWidth = LOWORD(dw);

    /* Определяет ширину прямоугольников */
    xStart = xText + cxWidth;
    xEnd = r.right - 10;

    /* Цикл по всем кистям */
    for (i=0; i<BRUSHCOUNT; i++, yText += yIncr)
    {
        TextOut (hdc, xText, yText, apszDesc[i], strlen(apszDesc[i]));
        SelectObject (hdc, hbr[i]);
        yLine = yText + yIncr - tm.tmHeight / 4;
        Rectangle (hdc, xStart, yText, xEnd, yLine);
    }
}

```

BRUSHES.RC

```

snapshot icon Brushes.ico
hand cursor Brushes.cur
square bitmap square.bmp

```

BRUSHES.DEF

NAME BRUSHES

EXETYPE WINDOWS

DESCRIPTION 'Brushes - Sample GDI Brushes'

CODE MOVEABLE DISCARDABLE

DATA MOVEABLE MULTIPLE

HEAPSIZE 512

STACKSIZE 5120

Программа Brushes выполняет свою работу в ответ на три события: создание окна, его уничтожение и рисование. В терминах языка C++ наш класс оконных объек-

тов имеет три компонентные функции: конструктор, деструктор и функцию для обработки сообщения WM_PAINT.

Во время своего создания окно определяет необходимые ему кисти GDI. Обычно оконный объект создает кисти GDI в своем конструкторе и сохраняет их в течение жизни окна. И хотя кисти занимают оперативную память, этот подход обеспечивает более быструю обработку WM_PAINT.

Если Вы создаете объект GDI, то должны его "убрать" по завершении работы. Поскольку мы создали кисти в конструкторе, имеет смысл уничтожить кисти в деструкторе нашего оконного объекта. Если Вы не будете следить за объектами GDI, то существует риск потерять системную память.

В ответ на сообщение WM_PAINT функция - обработчик сообщений, устанавливаемая по умолчанию, вызывает компонентную функцию Paint нашего оконного объекта. Эта функция разделяет доступную "клиентную" область на 11 частей. Каждая часть представляет собой прямоугольник, который был нарисован своей кистью.

В этой программе мы вызываем три текстовые подпрограммы GDI: GetTextMetrics, GetTextExtent и TextOut. Подпрограмма GetTextMetrics выдает информацию о размерах текущего шрифта, или таблицу графических шаблонов текста. Подпрограмма GetTextEvent применяется для определения ширины и высоты некоторой строки текста, а TextOut выводит строку текста на экран.

В следующей главе мы ближе познакомимся с этими подпрограммами и другими средствами GDI для вывода текста.

Глава 10. Вывод текста

На выходе большинства программ появляется именно текст. Однако мы отложили обсуждение текста вплоть до настоящей главы, так как его вывод в GDI существенно отличается от вывода в традиционной среде программирования. GDI рассматривает текст как тип графического объекта.

Может быть, Вам доводилось писать программы со строкоориентированным выводом? Этот метод был сначала применен в первых интерактивных вычислительных системах, в которых для ввода и вывода информации использовалась пишущая машинка. На экранах современных дисплеев при этом типе вывода строки текста "сдвигаются" за пределы экрана в воображаемую мусорную корзину. Диалог очень простой: компьютер выводит на экран запрос, а пользователь отвечает. Начинающие программисты, работающие на языке C всегда используют этот тип вывода, потому что он является частью стандартной библиотеки, и пишут примерно так:

```
printf ("Введите первое число:");  
scanf ("%i", &iValue);
```

Затем появился экранно-ориентированный вывод, который представляет экран дисплея как сетку знакомест. Этот метод обычно используют программы типа систем подготовки текстов и полноэкранных редакторов. Язык dBase, один из популярных языков баз данных, работающих под управлением MS DOS, содержит подобные команды для вывода текста и ввода информации на экран дисплея:

```
@ 10, 25 SAY "Пожалуйста, введите Ваше имя:"  
@ 10, 49 GET NAME
```

Подход GDI к выводу текста может быть описан как "пиксел-ориентированный" вывод, поскольку GDI не имеет представления о размере знакоместа. GDI позволяет размещать текст с помощью той же сетки пикселей, которую Вы применяете для изображения линий, прямоугольников и других геометрических фигур. Это дает возможность хорошо управлять размещением текста и легко соединять его с геометрическими фигурами. Так же легко Вы сможете соединять даже тексты с различными размерами и стилями.

В отличие от других графических объектов, текст не выводится с помощью обычных уравнений. Для его вывода требуются шрифты. Шрифт - это база данных графических шаблонов, которые описывают форму и размер каждой буквы, цифры и знака препинания. Все устройства GDI поддерживают один или несколько шрифтов. Ниже представлены основные шрифты GDI.

Каждый драйвер дисплея GDI содержит набор основных шрифтов, похожих на основные шрифты адаптера дисплея VGA. Обычный набор основных шрифтов обеспечивает минимальный уровень поддержки для всех дисплеев.

Фирма Microsoft тщательно отобрала следующие шрифты как часть основного набора (рис.10.1):

Courier - обеспечивает шрифт с постоянным шагом, как на пишущей машинке.

Tms Rmn - набор пропорциональных шрифтов с засечками в широком диапазоне размеров. Этот набор может быть использован для замены других шрифтов с засечками в программах, которые имитируют на экране дисплея выдаваемые принтером символы.

Helv - набор пропорциональных рубленых шрифтов типа Helvetica, также в широком диапазоне размеров. Подобно шрифту Tms Rmn, эти шрифты предназначены для использования в программах, которые имитируют на экране дисплея выдаваемые принтером символы.

Symbol - альтернативный набор символов, который может быть использован в качестве шрифта, этот шрифт содержит буквы греческого алфавита.

Roman, Modern u Script - штриховые шрифты, в которых графические шаблоны символов представлены в виде совокупности штрихов или отрезков прямых.

Набор имеющихся в распоряжении шрифтов может быть расширен и за пределы основного набора. Конечный пользователь может купить шрифты и установить их с помощью Control Panel. Программисты могут создавать собственные шрифты и включать их в программы для Windows. Фирма Microsoft включила новый тип шрифтов в версию 3.1 Windows с использованием "механизма отображения шрифтов" TrueType.

Процесс размещения текста часто включает в себя определение метрики шрифта, т.е. таблицы с информацией о размерах конкретного шрифта.

Если Вы хотите представлять буфер вывода в виде матрицы из 25 строк и 80 столбцов, GDI обеспечит Вас инструментальными программными средствами. Однако Вы должны будете уделить внимание и другим моментам, таким как размер буфера вывода (окна могут сжиматься и увеличиваться) и размер текста (каждому дисплею доступны более 20 размеров и стилей для выбора). И если Вы хотите, чтобы Ваша программа выполнялась на любых дисплеях и с любыми принтерами, Вы должны сделать ее не зависящей от внешних устройств.

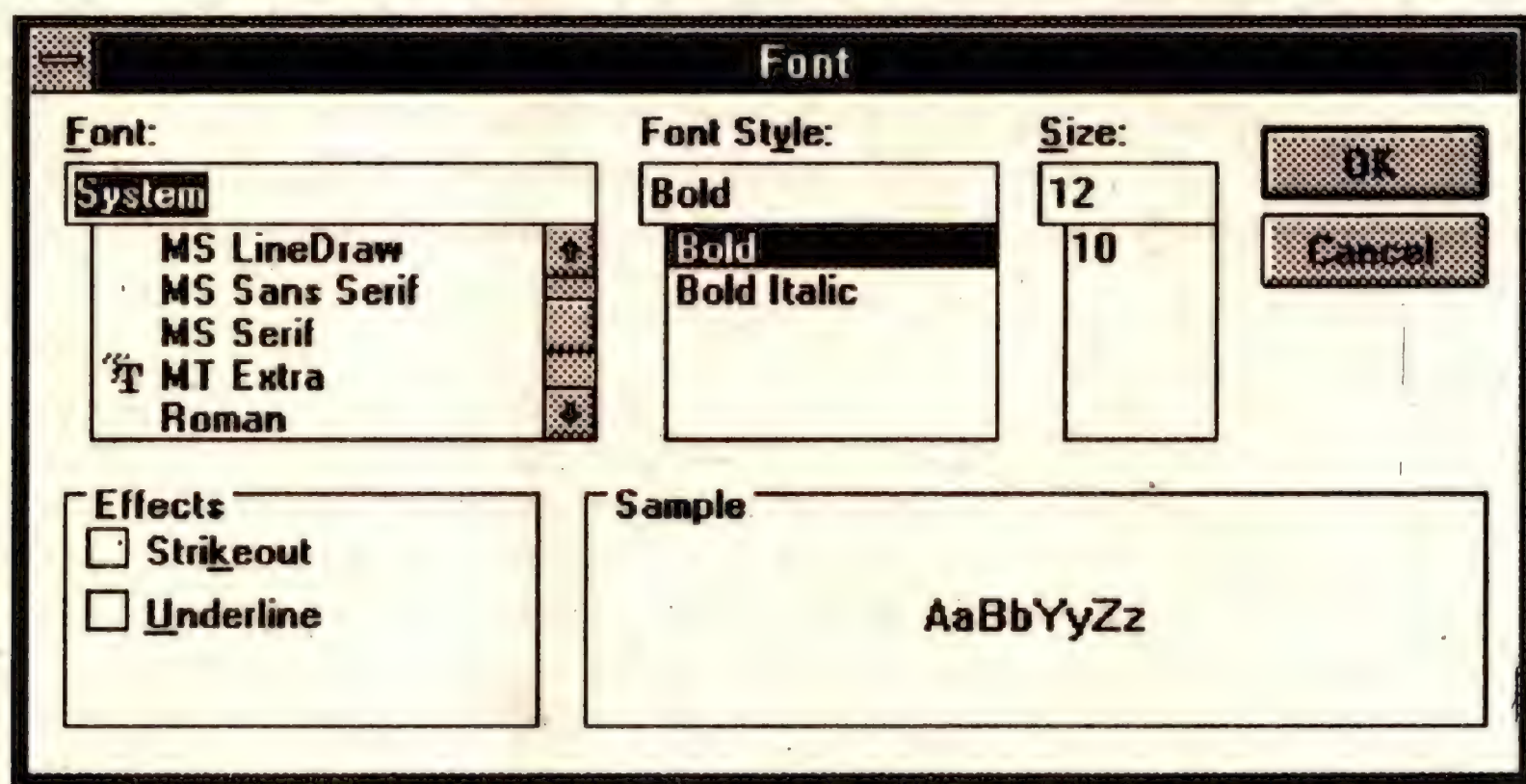


Рис.10.1. Примеры основных наборов шрифтов для монитора VGA.

Хотя на первых порах может показаться странной необходимость размещать текст в сетке пикселей, без этого будет невозможно соединять текст с графическими объектами других типов. В GDI текст размещается с помощью той же пиксел-ориентированной сетки, которую мы использовали для рисования графических маркеров, линий и прямоугольников.

Отметим некоторые ключевые вопросы, которые мы собираемся рассмотреть в этой главе. Начнем с обзора доступных текстовых подпрограмм. Затем остановимся на атрибутах контекста устройства, которые имеют отношение к тексту, и уделим особое внимание главному из них - шрифту. А закончим главу обсуждением метрики текста, играющей важную роль при создании текстового вывода, не зависящего от внешних устройств.

Примитивы для вывода текста

Для вывода текста имеется пять подпрограмм: DrawText, ExtTextOut, GrayString и TabbedTextOut. Три из них, DrawText, GrayString и TabbedTextOut, являются частью не GDI, а Менеджера Окон, тем не менее, они обеспечивают расширение возможностей GDI.

Давайте сначала поговорим о TextOut, вероятно, наиболее широко используемой текстовой подпрограмме GDI.

Подпрограмма TextOut

Подпрограмма TextOut - это одна из самых простых текстовых подпрограмм GDI: она выводит одну строку текста. Мы уже использовали эту программу для вывода ярлыков в одной из первых программ, но теперь познакомимся с ней поближе. Подпрограмма TextOut определена следующим образом:

TextOut (hDC, x, y, lpString, nCount)

* hDC - логический номер контекста устройства. Он сообщает GDI об устройстве для вывода и об используемых атрибутах для рисования.

* x и y - целые числа, определяющие точку привязки при размещении текста. Точка привязки - это положение в координатной системе, определенной в контексте устройства. А поскольку мы ограничились координатной системой MM_TEXT, то нашими элементами будут пиксели.

* lpString - дальний указатель на символьную строку. Она не должна быть строкой, заканчивающейся пустым символом, так как TextOut получает длину строки из параметра nCount.

* nCount - число символов в строке текста.

По умолчанию GDI размещает строку текста в верхнем левом углу в точке привязки. Позднее в этой главе мы поговорим об атрибутах контекста устройства, опишем другие возможности выравнивания текста.

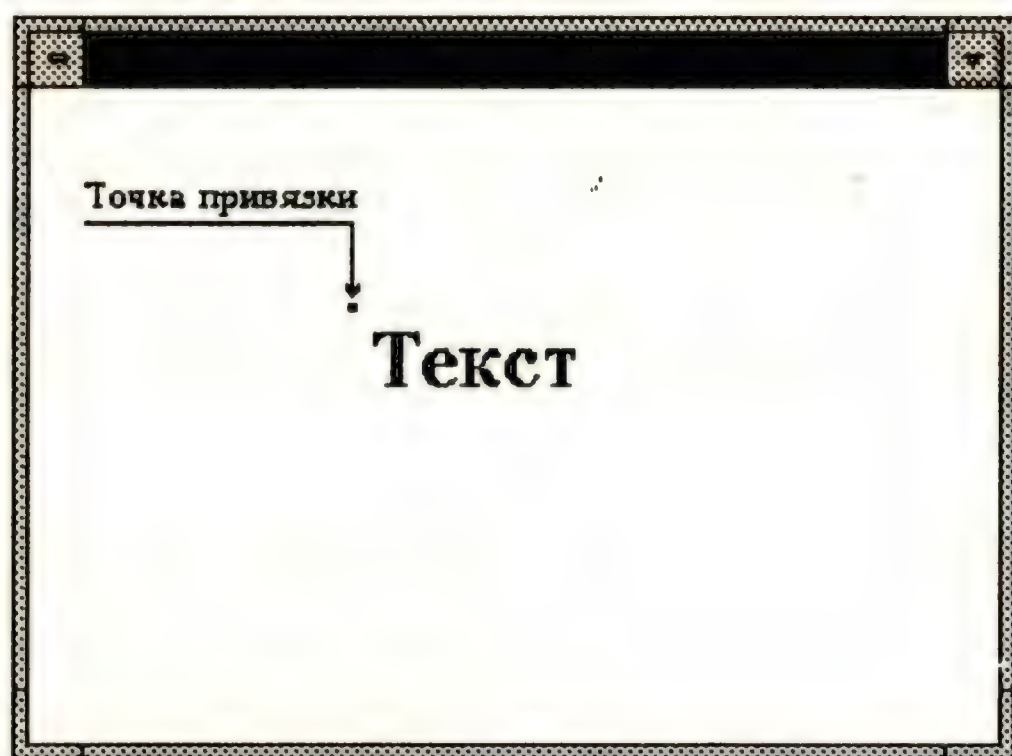


Рис.10.2. Взаимное расположение точки привязки и текста, используемое по умолчанию.

Вызывать TextOut удобно, как правило, следующим образом:

```
static xchar ac[] = "Вывести на экран этот текст.";
BeginPaint (hwnd, &ps);
TextOut (ps.hdc, X, Y, ac, strlen(ac));
EndPaint (hwnd, &ps);
```

В этом коде длина строки вычисляется "на лету" - с помощью strlen, подпрограммы для Windows, аналога стандартной функции strlen.

Некоторые программисты предпочитают разделять процедуру определения длины строки и ее вывода:

```
static xchar ac[] = "Вывести на экран этот текст.";
static int cb = sizeof(ac) - 1;
BeginPaint (hwnd, &ps);
TextOut (ps.hdc, X, Y, ac, cb);
EndPaint (hwnd, &ps);
```


При этом длина строки текста определяется во время компиляции, а не во время выполнения. В результате ускоряется процесс рисования. Сэкономленное таким образом время может способствовать повышению эффективности работы программы.

Следующая текстовая подпрограмма напоминает TextOut, но обеспечивает несколько дополнительных возможностей.

Подпрограмма ExtTextOut

Подобно TextOut, подпрограмма ExtTextOut рисует (выводит) одну строку текста, но у нее есть три дополнительных параметра: управление интервалами между символами, вырезка и непрозрачный прямоугольник. В соответствии со своими требованиями Вы можете соединять и подбирать эти дополнительные возможности (см. ниже). Подпрограмма ExtTextOut определена следующим образом:

ExtTextOut (hDC, X, Y, wOptions, lpRect, lpString, nCount, lpDx);

* hDC - логический номер контекста устройства. Он сообщает GDI об устройстве для вывода и об используемых атрибутах для рисования.

* X и Y - целые числа, определяющие точку привязки при размещении текста. Точка привязки - это положение в координатной системе, определенной в контексте устройства. Поскольку мы ограничились координатной системой MM_TEXT, то нашими элементами будут пиксели.

* wOptions - флаг для двух или трех дополнительных возможностей, присущих в этой подпрограмме. Флаг может иметь как значения 0, ETO_CLIPPED, ETO_OPAQUE, так и комбинацию значений ENT_CLIPPED | ETO_OPAQUE. Объяснение приведено ниже.

* lpRect - указатель на прямоугольник. В зависимости от значения wOptions параметр lpRect может указывать на вырезанный, непрозрачный или вырезанный непрозрачный прямоугольник.

* lpString - дальний указатель на символьную строку. Она не должна быть строкой, заканчивающейся пустым символом, так как TextOut получает длину строки из параметра nCount.

* nCount - число символов в строке текста.

* lpDx указывает на массив значений интервалов между символами.

Вначале обсудим возможность задания интервала между символами. В отличие от подпрограммы TextOut, которая использует значение интервала между символами, заданное по умолчанию в шрифте, подпрограмма ExtTextOut позволяет определить свои собственные значения интервала между всеми символами. Вы задаете значения интервалов в массиве целых чисел. Последний параметр подпрограммы lpDx является указателем на этот массив.

Ниже приведен исходный текст примера программы Exttext:

MAKEFILE.MAK

.AUTODEPEND

Определения транслятора

INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;

C:\BORLANDC\INCLUDE

CC = bcc -c -D_CLASSDLL -H -ml -WS -w -I\$(INC)

Неявные правила

.c.obj:

\$(CC) {S< }

.cpp.obj:

\$(CC) {S< }

Явные правила

ExtText.exe: ExtText.res ExtText.def ExtText.obj

tlink /c/C/n/P-/Twe/x @ExtText.LNK

RC ExtText.res ExtText.exe

Зависимости отдельных файлов

ExtText.obj: ExtText.cpp.

ExtText.res: ExtText.rc ExtText.cur ExtText.ico

RC -R -FO ExtText.res ExtText.RC

EXTTEXT.LNK

c:\borlandc\lib\cOwl.obj+
ExtText.obj
ExtText, ExtText
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwl.lib+
import.lib+
mathl.lib+
cl.lib
ExtText.def

EXTTEXT.CPP

/*

EXTTEXT.CPP Демонстрационный пример для программы ExtTextOut.

*/

#include <owl.h>

/*

Объявления классов

*/

class TExtTextApplication : public TApplication

{

public:

TExtTextApplication (LPSTR lpszName, HANDLE hInstance,

HANDLE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow);

virtual void InitMainWindow ();

};

class TExtTextWindow : public TWindow

{

public:

TExtTextWindow (PTWindowsObject pwParent,

LPSTR lpszTitle, PTModule pmModule);

virtual LPSTR GetClassName ();

virtual void GetWindowClass (WNDCLASS&);

virtual void Paint(HDC hdc, PAINTSTRUCT& ps);

};

/*

Прототипы функций

*/

void NEAR PASCAL ExtTextClipping(HDC hdc);

void NEAR PASCAL ExtTextOpaqueRect(HDC hdc, HWND hwnd);

void NEAR PASCAL ExtTextSpacing(HDC hdc);

/*

Основная функция: WinMain

*/

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
LPSTR lpszCmdLine, int nCmdShow)


```

{
    TExtTextApplication ExtText ("ExtText", hInstance, hPrevInstance,
                                lpszCmdLine, nCmdShow);
    ExtText.Run();
    return ExtText.Status;
}

/*
Компонентная функция класса прикладных программ
*/
TExtTextApplication::TExtTextApplication (LPSTR lpszName,
                                           HANDLE hInstance, HANDLE hPrevIntance,
                                           LPSTR lpszCmdLine, int nCmdShow)
    :TApplication (lpszName, hInstance,
                  hPrevInstance, lpszCmdLine, nCmdShow)
{
    /* Здесь инициализируется прикладная программа */
}

/*
Компонентная функция класса прикладных программ
*/
void TExtTextApplication::InitMainWindow ()
{
    MainWindow = new TExtTextWindow(NULL,
                                    "Three Features of ExtTextOut", NULL);
}

/*
Компонентная функция класса TExtTextWind
*/
TExtTextWindow::TExtTextWindow (PTWindowsObject pwParent,
                                LPSTR lpszTitle, PTModule pmModule)
    :TWindow(pwParent, lpszTitle, pmModule)
{
    /* Здесь инициализируется окно */
}

/*
Компонентная функция класса TExtTextWind
*/
LPSTR TExtTextWindow::GetClassName ()
{
    return "ExtText:MAIN";
}

/*
Компонентная функция класса TExtTextWind
*/
void TExtTextWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
}

```



```

/*
Компонентная функция класса TExtTextWind
*/
void TExtTextWindow::Paint(HDC hdc, PAINTSTRUCT& ps)
{
    ExtTextSpacing(hdc);
    ExtTextClipping(hdc);
    ExtTextOpaqueRect(hdc, HWindow);
}

/*
ExtTextClipping: Демонстрационный пример вырезки с помощью ExtTextOut
*/
void NEAR PASCAL ExtTextClipping (HDC hdc)
{
    static char ach[] = "Clipping";
    static int cch = sizeof (ach) - 1;
    int x;
    int y;
    int yHeight;
    RECT r;
    TEXTMETRIC tm;

    GetTextMetrics (hdc, &tm);
    yHeight = tm.tmHeight + tm.tmExternalLeading;
    x = tm.tmAveCharWidth * 5;
    y = yHeight * 6;

    r.top = y;
    r.left = x + 20;
    r.right = x + 90;
    r.bottom = y + tm.tmHeight * 8 + tm.tmHeight / 2;
    Rectangle (hdc, r.left-1, r.top-1, r.right+1, r.bottom+1);
    while (y < r.bottom)
    {
        ExtTextOut(hdc, x, y, ETO_CLIPPED, &r, ach, cch, NULL);
        y += yHeight;
        x += 8;
    }
}

/*
ExtTextOpaqueRect: Демонстрационный пример построения
с помощью ExtTextOut непрозрачного прямоугольника
*/
void NEAR PASCAL ExtTextOpaqueRect(HDC hdc, HWND nwcc)
{
    static char ach[] = "Drawing an Opaque Rectangle";
    static int cch = sizeof (ach) - 1;
    int x;
    int y;
    int yHeight;
    RECT r;
    TEXTMETRIC tm;

    GetTextMetrics (hdc, &tm);
    yHeight = tm.tmHeight + tm.tmExternalLeading;
    x = tm.tmAveCharWidth * 5;

```



```

y = yHeight * 15;
GetClientRect (nwcc, &r);
r.top = y;
r.bottom = r.top * (5 + tm.tmHeight);
r.left = x;
r.right = x + tm.tmAveCharWidth * 50;
SetBkColor (hdc, RGB (0, 0, 0));
SetTextColor (hdc, RGB (255, 255, 255));
SetTextAlign (hdc, TA_CENTER | TA_BASELINE);
x = (r.left + r.right) / 2;
y = (r.top + r.bottom) / 2;
ExtTextOut (hdc, x, y, ETO_OPAQUE, &r, ach, cch, NULL);
}

```

/*

ExtTextSpacing: Изменение с помощью ExtTextOut расстояния между символами

*/

```

void NEAR PASCAL ExtTextSpacing (HDC hdc)
{
    static char acl[] = "Character Spacing";
    static int cbl = sizeof (acl) - 1;
    static int ai1[] = {6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6};
    static int ai2[] = {9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9};
    static int ai3[] = {12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12};

    int x;
    int y;
    int yHeight;
    RECT r;
    TEXTMETRIC tm;
    GetTextMetrics (hdc, &tm);
    yHeight = tm.tmHeight + tm.tmExternalLeading;
    x = tm.tmAveCharWidth * 5;
    y = yHeight;
    ExtTextOut (hdc, x, y, 0, NULL, acl, cbl, ai1);
    y += yHeight;
    ExtTextOut (hdc, x, y, 0, NULL, acl, cbl, ai2);
    y += yHeight;
    ExtTextOut (hdc, x, y, 0, NULL, acl, cbl, ai3);
    y += yHeight;
}

```

EXTTEXT.RC

snapshot icon ExtText.ico
hand cursor ExtText.cur

EXTTEXT.DEF

NAME EXTTEXT

EXETYPE WINDOWS

.DESCRIPTION 'ExtText - ExtTextOut Demo'

CODE MOVEABLE DISCARDABLE

DATA MOVEABLE MULTIPLE

HEAPSIZE 512

STACKSIZE 5120

В программе Exttext три подпрограммы - ExtTxtSpacing, ExtTxtClipping и ExtTxtOpaqueRect - демонстрируют различные возможности ExtTextOut.

Программа выводит на экран строку "Character spacing", используя массив с тремя различными интервалами между символами: 6, 9 и 12 пикселей. Длина третьей строки (с интервалом в 12 пикселей) в два раза превышает длину первой (с интервалом в 6 пикселей). Эта особенность используется для выравнивания текста путем расширения его границ до левого и правого полей. Например, текст на этой странице выровненный. В зависимости от значений в массиве интервалов между символами мы можем подогнать строку текста к любому краю. Вот текст подпрограммы из нашего примера:

```
/*
ExtTextSpacing: Демонстрация интервалов между символами
подпрограммы ExtTextOut.
*/
void NEAR PASCAL ExtTextSpacing(HDC hdc)
{
    static char acl[] = "Character Spacing";
    /* "Интервал между символами" */
    static int cbl = sizeof(acl) - 1;
    static int ai1[] = (6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6);
    static int ai2[] = (9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9);
    static int ai3[] = (12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12);
    int x;
    int y;
    int yHeight;
    TEXTMETRIC tm;

    GetTextMetrics(hdc, &tm);
    yHeight = tm.tmHeight + tm.tmExternalLeading;
    x = tm.tmAveCharWidth * 5;
    y = yHeight;
    ExtTextOut(hdc, x, y, 0, NULL, acl, cbl, ai1);
    y += yHeight;
    ExtTextOut(hdc, x, y, 0, NULL, acl, cbl, ai2);
    y += yHeight;
    ExtTextOut(hdc, x, y, 0, NULL, acl, cbl, ai3);
    y += yHeight;
}
```

Наш массив символов содержит 17 символов, это означает, что имеется 16 интервалов между буквами, следовательно, наш массив интервалов между символами имеет 16 значений. Каждое значение определяет ширину одного знакоместа. Хотя для заполнения массива интервалов между символами требуются некоторые дополнительные усилия, Вы получаете возможность контролировать положение каждой буквы в строке текста.

Обратите внимание на вызов GetTextMetrics, который обеспечивает информацию о размерах для текущего шрифта. Мы используем значения, возвращаемые этой подпрограммой, для размещения первой строки текста с отступлением на одну строку от верхней границы окна и пять пробелов от левой границы окна. Более полно мы изучим эту подпрограмму дальше.

Вторая дополнительная возможность подпрограммы ExtTextOut заключается в том, что Вы можете определить вырезаемый прямоугольник для текста. Когда мы

вводили понятие контекста устройства, то отметили, что каждый контекст устройства имеет область вырезки. Функция ETO_CLIPPED подпрограммы ExtTextOut позволяет задать дополнительный вырезаемый прямоугольник. Кроме того, можно создать окно текста и обойтись без издержек, обусловленных созданием действительного окна. Вот текст подпрограммы из нашего примера, выполняющей вырезку:

```
/*
ExtTextClipping: Демонстрация вырезки в подпрограмме ExtTextOut.
*/
void NEAR PASCAL ExtTextClipping(HDC hdc)
{
    static char ach[] = "Вырезка изображения";
    static int cch = sizeof(ach) - 1;
    int x;
    int y;
    int yHeight;
    RECT r;
    TEXTMETRIC tm;

    GetTextMetrics(hdc, &tm);
    yHeight = tm.tmHeight + tm.tmExternalLeading;
    x = tm.tmAveCharWidth * 5;
    y = yHeight * 6;
    r.top = y;
    r.left = x + 20;
    r.right = x + 90;
    r.bottom = y + tm.tmHeight * 8 + tm.tmHeight / 2;
    Rectangle(hdc, r.left-1, r.top-1, r.right+1, r.bottom+1);
    while (y < r.bottom)
    {
        ExtTextOut(hdc, x, y, ETO_CLIPPED, &r, ach, cch, NULL);
        y += yHeight;
        x += 8;
    }
}
```

Мы полагаемся на подпрограмму GetTextMetrics, которая поможет нам выбрать место для вывода строк текста. Здесь отступаем на шесть строк от верхней границы окна и пять пробелов от левой границы окна.

Размеры вырезаемого прямоугольника были установлены произвольно и составили 70 пикселей в ширину и восемь с половиной строк в высоту. В нем используется координатная сетка с узлами на пересечениях, о которой впервые шла речь в главе 9, когда описывали процесс рисования заполненных областей.

Третья дополнительная возможность подпрограммы ExtTextOut состоит в создании непрозрачного прямоугольника. Благодаря этой функции можно стереть фоновую область при выводе строки текста. Как видно из примера программы, цвет фона выбирается при вызове SetBkColor, которая устанавливает значение атрибута контекста устройства, используемого всеми подпрограммами, выводящими (рисующими) текст. Мы остановимся более подробно на этих и других атрибутах контекста устройства.

Вот текст подпрограммы, создающей непрозрачный прямоугольник:

```
/*
ExtTextOpaqueRect: Демонстрация создания непрозрачного прямоугольника
в подпрограмме ExtTextOut.
*/
void NEAR PASCAL ExtTextOpaqueRect(HDC hdc, HWND hwnd)
{
```



```

static char ach[] = "Рисование непрозрачного прямоугольника";
static int cch = sizeof(ach) - 1;
int x;
int y;
int yHeight;
RECT r;
TEXTMETRIC tm;

GetTextMetrics(hdc, &tm);
yHeight = tm.tmHeight + tm.tmExternalLeading;
x = tm.tmAveCharWidth * 5;
y = yHeight * 15;
GetClientRect(hwnd, &r);
r.top = y;
r.bottom = r.top + (5 * tm.tmHeight);
r.left = x;
r.right = x + tm.tmAveCharWidth * 50;
SetBkColor(hdc, RGB(0, 0, 0));
SetTextColor(hdc, RGB(255, 255, 255));
SetTextAlign(hdc, TA_CENTER | TA_BASELINE);
x = (r.left + r.right) / 2;
y = (r.top + r.bottom) / 2;
ExtTextOut(hdc, x, y, ETO_OPAQUE, &r, ach, cch, NULL);
}

```

Подпрограмма `GetTextMetrics` поможет нам выбрать место для вывода текста и его черного непрозрачного фона. Здесь верхний левый угол прямоугольника расположен в 15 строках от верхней границы окна и в 5 пробелах от левой границы окна.

Мы установили размер прямоугольника следующим образом: 5 знакомест в высоту и 50 в ширину. Для того чтобы сделать непрозрачный прямоугольник видимым, мы изменили два атрибута контекста устройства: цвет для вывода текста (установлен белым) и цвет фона (установлен черным). Мы использовали третий атрибут контекста устройства - выравнивание текста для центрирования его внутри прямоугольника.

Подпрограмма `TabbedTextOut`

Третий примитив для вывода текста выводит строку текста, но при этом распространяет знаки табуляции до позиций табуляции. Этим обеспечивается удобный способ выравнивания столбцов данных. Фирма Microsoft включила эту подпрограмму в библиотеку Windows для блоков списка с целью облегчить создание списков данных, представленных в виде столбцов. Вы же можете просто использовать эту подпрограмму в своих программах для той же цели.

На рис.10.3. приведен пример текста, выведенного этой подпрограммой. Массив символьных строк этого примера:

```

#define COUNT 19
static char *apch[] = ("Страна \tСтолица",
    "----- \t-----",
    "Afghanistan \tKabul",
    "Albania \tTirana",
    "Algeria \tAlgiers",
    "Angola \tLuanda",
    "Antigua & Barbuda \tSt. John's",
    "Argentina \tBuenos Aires",
    "Australia \tCanberra",
    "Austria \tVienna",
    "The Bahamas \tNassau",
    "Bahrain \tManama",

```



```
"Bangladesh \tDhaka",
"Barbados \tBridgetown",
"Belgium \tBrussels",
"Belize \tBelmopan",
"Benin \tPorto-Nove",
"Bhutan \tThimphu",
"Bolivia \tLa Paz");
```

Страна	Столица
Afghanistan	Kabul
Albania	Tirana
Algeria	Algiers
Angola	Luanda
Antigua & Barbuda	St. John's
Argentina	Buenos Aires
Australia	Canberra
Austria	Vienna
The Bahamas	Nassau
Bahrain	Manama
Bangladesh	Dhaka
Barbados	Bridgetown
Belgium	Brussels
Belize	Belmopan
Benin	Porto-Nove
Bhutan	Thimphu
Bolivia	La Paz

Рис.10.3. Пример работы подпрограммы *TabbedTextOut*.

А вот текст функции *Paint* оконного объекта, с помощью которой был выведен приведенный выше текст:

```
void TSampleWindow ::Paint (HDC hdc, PAINTSTRUCT& ps)
{
    DWORD dwSize;
    int i;
    int xTab;
    int xText;
    int yText;
    int yHeight;

    dwSize = GetTextExtent (hdc, "X", 1);
    yHeight = yText = HIWORD(dwSize);
    xText = 3 * LOWORD (dwSize);
    xTab = 20 * LOWORD (dwSize);
    for (i=0;i<COUNT;i++, yText += yHeight)
    {
        TabbedTextOut (hdc, xText, yText, apch[i], strlen(apch[i]),
                        1, &xTab, xText);
    }
}
```

В этом тексте функция *GetTextExtent* вызывается для определения ширины и высоты буквы "X" в текущем (системном) шрифте. В предыдущих примерах для получения этой информации мы использовали подпрограмму *GetTextMetrics*, а сейчас для разнообразия применили этот альтернативный метод. Обратите внимание на ис-

пользование макропроцедур HIWORD и LOWORD, которые отфильтровывают желаемую часть значения возврата этой подпрограммы.

Мы используем результаты для вычисления трех значений: начальных позиций вывода текста X и Y (одна строка сверху и три символа слева) и положение позиции табуляции. В данном случае мы поместили позицию табуляции таким образом, чтобы она примерно на 20 пробелов отстояла от начальной точки (X и Y). Если бы было несколько позиций табуляции, нам пришлось бы для них создать массив. А сейчас мы просто объявили целое число и затем передали его подпрограмме.

Подпрограмма DrawText

Подобно TabbedTextOut подпрограмма DrawText обеспечивает некоторую возможность форматирования. По нашему мнению, наиболее полезная возможность заключается в автоматическом переходе на новую строку при большом количестве строк текста (хотя это не единственное достоинство DrawText).

На рис.10.4 показаны три результата выполнения примера программы, которая использует DrawText для вывода длинной строки текста в областях с различными размером и формой. Массив символов записывается следующим образом:

```
static char *apchDesc = "The DT_WORDBREAK flag makes "  
    "DrawText split a long string "  
    "into several lines of text. "  
    "As you can tell, the split is "  
    "only perfomed at normal word "  
    "breaks.";
```

/* Флаг DT_WORDBREAK заставляет DrawText разбить длинную строку
на несколько строк текста. Как Вы видите, разбивка выполняется
только по естественным разрывам в тексте. */

Компонентная функция Paint оконного объекта может быть записана следующим образом:

```
void TSampleWindow::Paint (HDC hdc, PAINTSTRUCT& ps)  
{  
    DWORD dwSize;  
    RECT r;  
  
    dwSize = GetTextExtent (hdc, "X", 1);  
    GetClientRect (hwnd, &r);  
    r.top    += HIWORD (dwSize);  
    r.bottom -= HIWORD (dwSize);  
    r.left   += LOWORD (dwSize) * 2;  
    r.right  -= LOWORD (dwSize) * 2;  
    DrawText (hdc, apchDesc, lstrlen (apchDesc), &r, DT_WORDBREAK);  
}
```

Напомним, что мы используем подпрограмму GetTextExtent для определения ширины и высоты буквы "X". Мы используем эти значения для смещения внутри "клиентной" области таким образом, чтобы около текста оставались поля.

Подпрограмма GetClientRect возвращает координаты "клиентной" области в переменной RECT. Затем мы изменяем значения в этой структуре так, чтобы создать поля со всех четырех сторон текста. Полученный в результате прямоугольник передается подпрограмме DrawText, которая использует эти значения для определения места вывода текста.

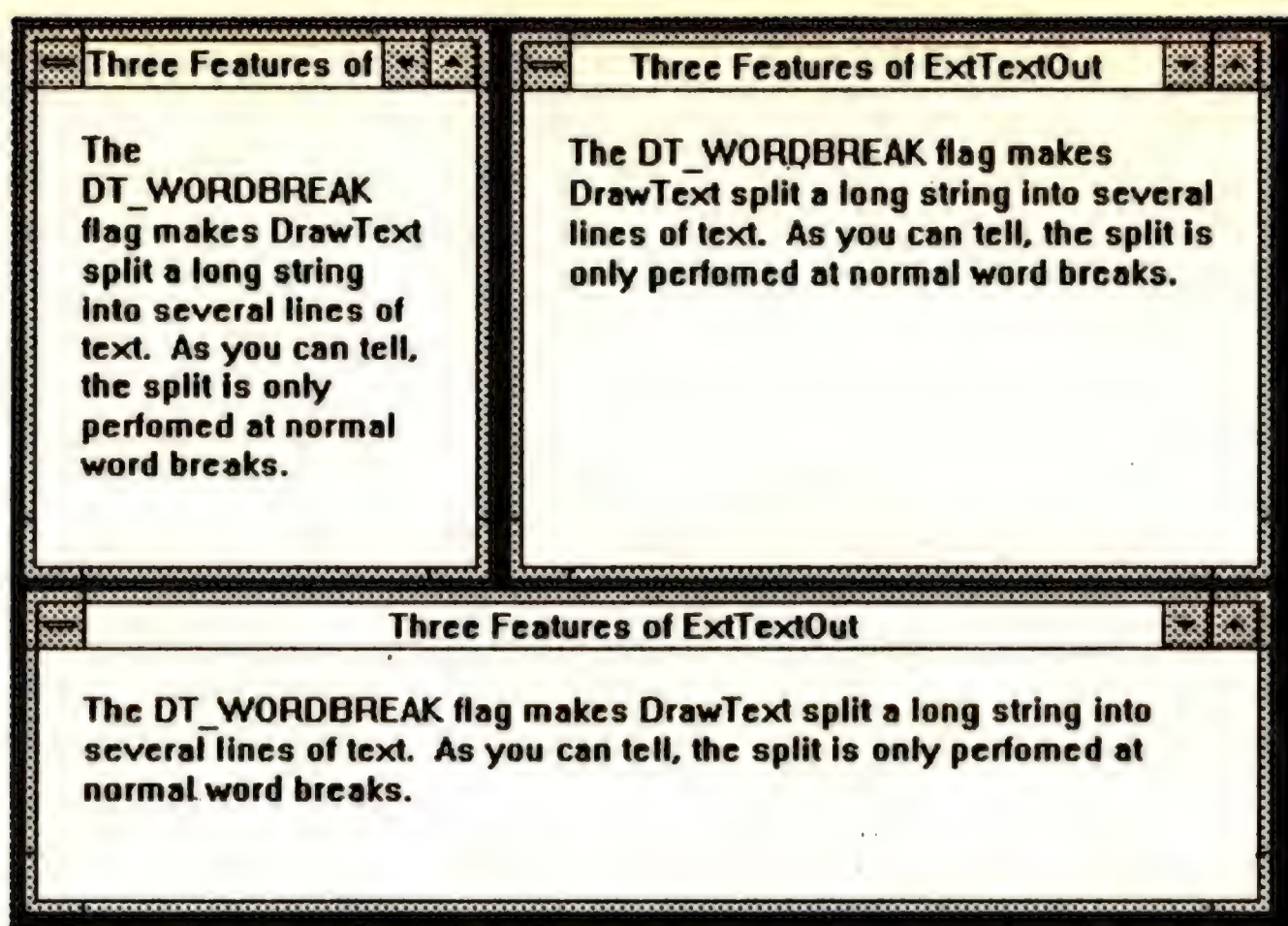


Рис.10.4. Пример работы подпрограммы DrawText.

Подпрограмма GrayString

Текст, выводимый этой подпрограммой, лучше всего описывается как "изменяемый". Из имени же подпрограммы (GrayString - серая, неяркая строка) видно, зачем фирма Microsoft ее создала: для вывода неяркого текста.

Менеджер окон использует эту подпрограмму для отображения заблокированных пунктов меню и заблокированных элементов управления блоками диалога. Если Вы пишете свои собственные элементы управления блоком диалога (обсуждаемые в главе 14) или создаете пользовательские пункты меню (обсуждаемые в главе 12), то эта подпрограмма может быть для Вас очень полезна.



Рис.10.5. Пример работы подпрограммы GrayString.

Вот определения для вывода строки текста:

```
static char acString[] = "Это неяркая строка";
static int cb = sizeof (acString) - 1;
```

А это текст компонентной функции Paint оконного объекта для вывода текста:

```
void TSampleWindow::Paint (HDC hdc, PAINTSTRUCT& ps)
{
    DWORD dwSize;
    int xText, yText;
    TEXTMETRIC tm;

    GetTextMetrics (hdc, &tm);
    xText = tm.tmAveCharWidth * 3;
    yText = tm.tmHeight * 2;
    GrayText (hdc, Object (BLACK BRUSH),
        NULL, (DWORD)(LPSTR)acString, cb, xText, yText, 0, 0);
}
```


В предыдущих примерах при размещении текста обращением к `GetTextMetrics` обеспечивалась информация о размерах. В этом - мы помещаем неяркою строку, отступив на две строки сверху и сделав три интервала между символами слева.

Это единственная подпрограмма вывода текста, которая использует кисть. Цвет кисти определяет цвет фона "изменчивого" текста. В этом примере мы используем черную кисть, что обеспечивает неяркое представление. Если бы мы выбрали красную кисть, на экране появились бы "красно-белые светотени".

На этом мы заканчиваем обзор различных подпрограмм GDI вывода текста. Как Вы могли видеть, вызов каждой из этих подпрограмм приводит к небольшому количеству различных результатов. Но библиотечные подпрограммы - это только полдела. Для получения полной картины того множества элементов управления, которыми обеспечивает GDI для вывода текста, нам необходимо обратиться к информации об атрибутах контекста устройства, которые влияют на представление текста.

Атрибуты контекста устройства для вывода текста

На представление и размещение текста влияют следующие шесть атрибутов:

Атрибут	Описание
Цвет фона	Цвет "пустого пространства" текста
Режим фона	Включение/выключение цвета фона
Шрифт	Стиль и размер текста
Задание интервалов	Дополнительные пиксели между символами для выравнивания текста
Выравнивание текста	Соотношение текста и опорной точки
Цвет для вывода	Цвет самих букв текста

Наиболее важным из перечисленных атрибутов является шрифт, определяющий форму и размер каждого символа. Поговорим о том, как GDI обрабатывает шрифты. Но сначала исследуем некоторые другие атрибуты текста, и начнем с управляющих цветом.

Цвет

Три различных атрибута контекста устройства имеют дело с цветом для вывода текста: цвет для вывода текста, цвет фона и режим фона. Текст может быть выведен только "чистым" цветом, а не размытым, как при заполнении областей. Наряду с пикселем, пером и цветами кисти Вы можете для определения цветов применять любой из трех методов: тройку RGB, индекс палитры или относительный номер в палитре RGB.

Атрибуты цвета для вывода текста определяют фактический цвет букв. Если бы для создания букв на этой странице использовался GDI, цвет для вывода текста нужно было бы установить черным. Для этого нужно вызвать процедуру `SetTextColor`, которая имеет следующие параметры:

`SetTextColor (hDC, crColor);`

- * `hDC` - логический номер контекста устройства. Он сообщает GDI об устройстве для вывода и используемых атрибутах рисования.
- * `crColor` - это желаемый цвет для вывода текста.

Параметр `crColor` - это исходное значение цвета для использования в одном из трех методов. Вот как установить голубой цвет для вывода текста с использованием тройки RGB:

`SetTextColor (hDC, RGB (0, 0, 0xFF));`

Атрибут цвета фона определяет цвет тех областей внутри знакоместа, которых не касается цвет для вывода текста. С точки зрения GDI текст на этой странице имеет белый фон, поскольку он является цветом пространства между буквами и цветом областей внутри таких букв, как "О" и "Q".

Как Вы могли заметить, атрибут цвета фона используется и для штрихующих кистей, стилизованных линий при установке соответственно цвета между штрихами и пустой области внутри шаблона стилизованной линии.

Для установки цвета фона нужно вызвать SetBkColor, которая принимает те же два параметра, что и SetTextColor:

SetBkColor (hDC, crColor);

- * hdc - логический номер контекста устройства. Он сообщает GDI об устройстве для вывода и используемых атрибутах рисования.
- * crColor - это желаемый цвет фона.

Вот пример установки зеленого фона:

SetBkColor (hDC, RGB (0, 0xFF, 0));

Третий атрибут цвета, режим фона - это переключатель для цвета фона. Подпрограмма SetBkMode, управляющая этим атрибутом, принимает два параметра:

SetBkMode (hDC, nBkMode);

- * hdc - логический номер контекста устройства. Он сообщает GDI об устройстве для вывода и используемых атрибутах рисования.
- * crColor - желаемый режим фона, принимающий значения OPAQUE (непрозрачный) или TRANSPARENT (прозрачный).

Если параметр nBkMode принимает значение OPAQUE, то цвет фона включен, если - значение TRANSPARENT, то цвет фона выключен.

Для примера создадим три строки текста с использованием различных цветов фона и цветов для вывода текста. Обратите внимание на то, что вторую строку текста нельзя прочесть, так как оба цвета - цвет фона и цвет для вывода текста - установлены черными. Установлены следующие определения данных:

```
char acFirst[] = "Черный на Белом (по умолчанию)";  
char acSecond[] = "Черный на Черном (невидимый)";  
char acThird[] = "Белый на Черном (инвертированный)";
```

Ниже приведен текст подпрограммы Paint для вывода этих строк:

```
void TSampleWindow::Paint (HDC hdc, PAINTSTRUCT& ps)  
{  
    int X;  
    int Y;  
    TEXTMETRIC tm;  
  
    GetTextMetrics (hdc, &tm);  
    X = tm.tmAveCharWidth * 3;  
    Y = tm.tmHeight * 2;  
  
    TextOut (hdc, X, Y, acFirst, lstrlen (acFirst));  
    Y += tm.tmHeight * 2;  
  
    SetBkColor (hdc, RGB (0, 0, 0));  
    TextOut (hdc, X, Y, acSecond, lstrlen (acSecond));  
    Y += tm.tmHeight * 2;  
  
    SetTextColor (hdc, RGB (255, 255, 255));  
    TextOut (hdc, X, Y, acThird, lstrlen (acThird));  
}
```




Рис.10.6. Три различных сочетания цветов текста и фона.

В этом примере использованы только черный и белый цвета, но очень просто вывести красный или зеленый текст на устройстве, которое поддерживает цвет: задайте подходящее значение цвета, а GDI сделает все остальное.

Выравнивание текста

Атрибут выравнивания текста позволяет изменять соотношение между опорной точкой, которая представляет собой пару (X, Y), передающуюся в каждую подпрограмму, и выводимым текстом. На рис.10.7 показано девять различных способов выравнивания текста, при этом для наглядности в каждую опорную точку помещен маркер.

Для того чтобы выровнять текст, необходимо вызвать подпрограмму:

`SetTextAlign (hDC, wFlags);`

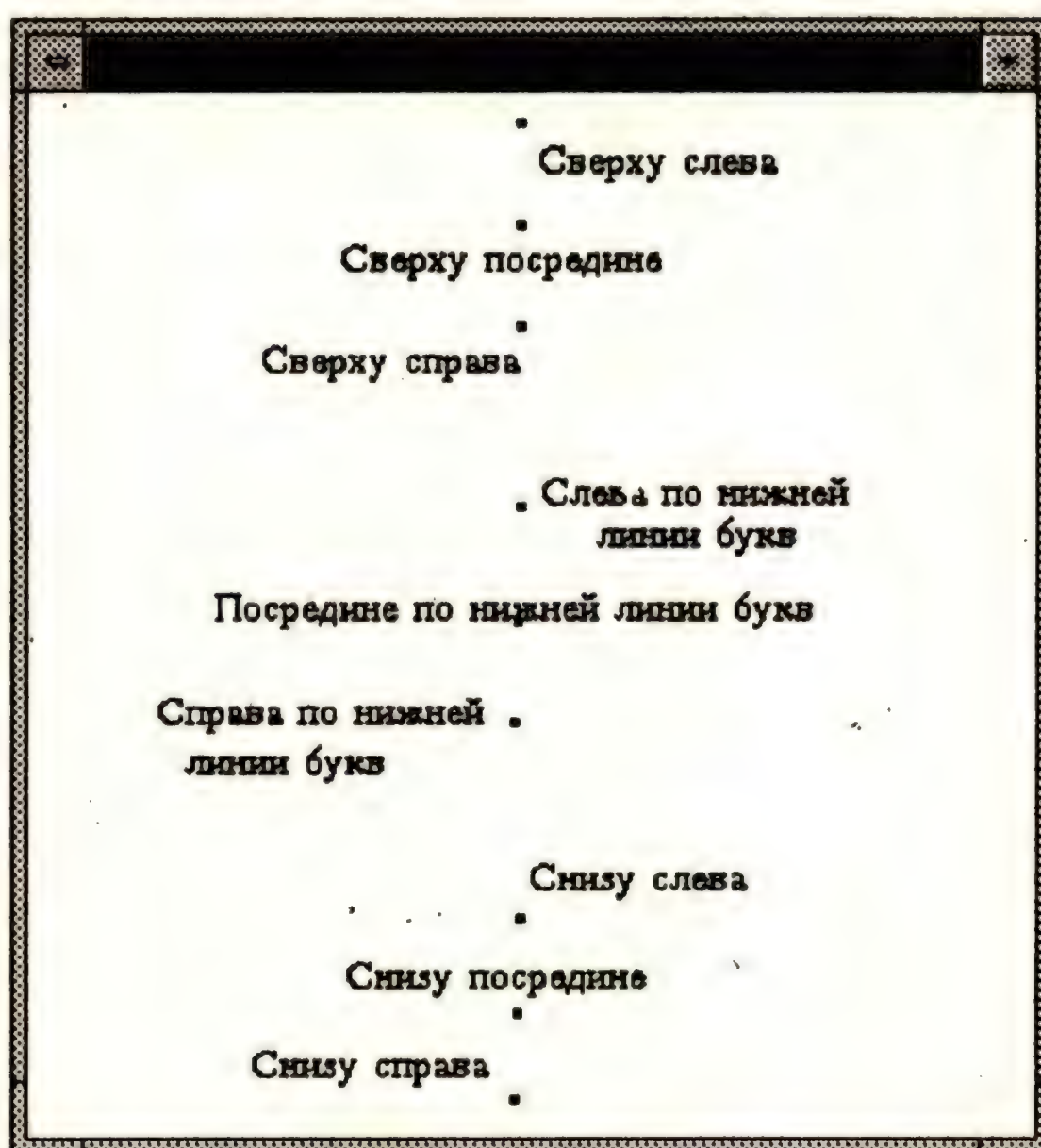


Рис.10.7. Девять различных способов выравнивания текста.

Как отмечено в примере, есть девять возможных способов выравнивания. Задать нужное выравнивание можно с помощью комбинаций двух флагов, выбирая значение каждого флага из соответствующего столбца приведенной ниже таблицы.

Горизонтальный флаг

TA_LEFT
TA_CENTER
TA_RIGHT

Вертикальный флаг

TA_TOP
TA_BASELINE
TA_BOTTOM

По умолчанию значение выравнивания - TA_LEFT | TA_TOP. Для выравнивания снизу и справа нужно задать:

```
SetTextAlign (hDC, TA_BOTTOM | TA_RIGHT);
```

Рис.10.7 был создан с помощью следующих фрагментов кода. Вот определения данных:

```
#define LINECOUNT 9
static char * apchDesc[LINECOUNT] =
( "Сверху в левом углу",
  "Сверху посередине",
  "Сверху в правом углу",
  "Слева по нижней линии",
  "Посередине нижней линии",
  "Справа по нижней линии",
  "Снизу в левом углу",
  "Снизу посередине",
  "Снизу в правом углу");
static int fAlign [LINECOUNT] =
{
  TA_LEFT | TA_TOP ,
  TA_CENTER | TA_TOP ,
  TA_RIGHT | TA_TOP ,
  TA_LEFT | TA_BASELINE ,
  TA_CENTER | TA_BASELINE ,
  TA_RIGHT | TA_BASELINE ,
  TA_LEFT | TA_BOTTOM ,
  TA_CENTER | TA_BOTTOM ,
  TA_RIGHT | TA_BOTTOM );
```

Вывод текста осуществлялся следующим образом:

```
void TSampleWindow::Paint (HDC hdc, PAINTSTRUCT& ps)
{
  int i;
  int xText;
  int yText;
  int yLineHeight;
  RECT r;

  GetClientRect (hwnd, &r);
  xText = r.right/2;
  yLineHeight = r.bottom / (LINECOUNT+1);
  yText = yLineHeight;

  for (i = 0; i < LINECOUNT; i++)
  {
    SetTextAlign (hdc, fAlign[i]);
    TextOut (hdc, xText, yText, apchDesc[i], lstrlen (apchDesc[i]));
  }
```



```

    DrawMarker (hdc, xText, yText);
    yText += yLineHeight;
}
}

```

В этом примере мы отступили от обычной практики задания интервалов между строками. Вместо использования подпрограмм `GetTextMetrics` или `GetTextExtent` мы разбили имеющееся пространство окна на 10 областей, по одной строке в области. Поскольку GDI рассматривает текст как графический объект, ничто не мешает Вам разрабатывать такие интересные и полезные методы, как этот, для использования экрана при выводе текста.

Задание интервалов между символами

Задание интервалов между символами позволяет Вам вставлять дополнительные пиксели между символами. Это обеспечивает другую возможность (помимо подпрограммы `ExtTextOut`) расширить строку текста для выравнивания ее по любому краю. Ниже приводится пример программы, которая выводит шесть строк текста с различными интервалами между символами - от 0 до 5 пикселей. Между соседними строками разницы почти нет, но верхняя и нижняя различаются сильно.



Рис.10.8. Демонстрация различных значений межсимвольного пространства.

Вначале нам нужны следующие элементы данных:

```

#define COUNT 6
static char acLine[] = "AaBbCcDdEeFfGgHhIiJjKkLlMm"
                        "NnOoPpQqRrSsTtUuVvWwXxYyZz";
static char *apch[] = ("0", "1", "2", "3", "4", "5");
static int  cbLine = sizeof (acLine) - 1;

```

Теперь текст `Paint` для вывода этих строк:

```

void TSampleWindow::Paint (HDC hdc, PAINTSTRUCT& ps)
{
    int i;
    int xText;
    int xText2;
    int yText;
    TEXTMETRIC tm;

    GetTextMetrics (hdc, &tm);
    xText = tm.tmAveCharWidth * 3;
    xText2 = tm.tmAveCharWidth * 6;
    yText = tm.tmHeight * 2;
    for (i = 0; i < COUNT; i++)
    {
        SetTextCharacterExtra (hdc, i);
        TextOut (hdc, xText, yText, apch[i], lstrlen (apch[i]));
    }
}

```



```

    TextOut (hdc, xText2, yText, acLine, cbLine);
    yText += tm.tmHeight;
}
}

```

Теперь опишем шрифт - еще один атрибут контекста устройства, который играет наиболее важную роль при определении формы и размера каждой буквы, выводимой GDI.

Шрифты

Шрифт - это совокупность шаблонов, используемых для вывода текста. Шрифты бывают всех форм, размеров и стилей. У шрифтов есть много общего с другими объектами рисования GDI, такими как перья и кисти. Прежде всего, шрифты могут совместно использоваться программами. И подобно другим объектам GDI, на шрифты ссылаются с помощью логического номера. Если программа готова к использованию конкретного шрифта, она выбирает логический номер шрифта и помещает его в контексте устройства, используя подпрограмму SelectObject:

```
SelectObject (hDC, hFont);
```

По сути дела, GDI распознает два типа шрифтов: логические и физические. Логический шрифт описывает текст стандартным не зависящим от внешних устройств способом. Как мы сейчас увидим, логический шрифт состоит из набора значений в структуре данных LOGFONT. Сам по себе логический шрифт не имеет достаточно информации для вывода текста на устройство. Подобно другим логическим объектам рисования, логический шрифт - это запрос на текст с определенными характеристиками.

Программа отображения шрифта GDI отбирает физический шрифт по описанию, содержащемуся в логическом шрифте. Отображение заканчивается, когда логический номер логического шрифта помещается в контексте устройства. Физический шрифт - это набор шаблонов, зависящий от устройства. Эти шаблоны используются для создания букв, цифр и знаков препинания, которые мы обычно связываем с текстом. Физический шрифт может быть аппаратно реализован, что типично для шрифтов принтера. GDI может держать шрифт в оперативной памяти.

Физические шрифты зависят от устройства, так как каждый шрифт создается в расчете на определенное устройство. Для подбора шрифтов к устройствам используются два измерения: разрешающая способность (определяемая числом пикселей на дюйм) и коэффициент сжатия. По умолчанию программа отображения шрифта GDI только отбирает физические шрифты, которые соответствуют метрике данного устройства. В противном случае результат будет непредсказуемым.

Например, разрешающая способность дисплея VGA составляет примерно 72 пикселя на дюйм. Современный стандартный лазерный принтер имеет 300 пикселей на дюйм. Если Вы попытаетесь сочетать шрифты на обоих устройствах, текст, распечатанный на лазерном принтере, будет неудобочитаемым из-за слишком мелких шрифтов VGA. И наоборот, если Вы используете шрифты лазерного принтера для создания текста в VGA, результирующий текст будет слишком крупным.

Если программе требуется точный контроль над выбором текста, она может запросить список физических шрифтов, поддерживаемых устройством. Этот процесс называется перечислением шрифтов. Если драйвер устройства перечисляет шрифты, он обеспечивает описание логического шрифта для каждого физического шрифта. Если программа хочет использовать конкретный шрифт, процесс идет в другую сторону: программа передает описание логического шрифта GDI, который, в свою очередь, обеспечивает связь с физическим шрифтом.

Многим программам не требуется серьезного контроля над представлением текста. Эти программы могут использовать шрифты по умолчанию, которые имеются в устройстве. Каждое устройство имеет такой шрифт. И неудивительно, что шрифт по умолчанию выбирается в контексте устройства.

На дисплее шрифт по умолчанию также называется системным шрифтом. Системы Windows 1.x и Windows 2.x использовали системный шрифт с постоянным шагом - тот, в котором все символы имеют одинаковую ширину. Начиная с Windows 3.0 системный шрифт является пропорциональным, т.е. некоторые символы шире остальных. Например, буква "W" занимает больше места, чем буква "i". Фирма Microsoft ввела это изменение, потому что текст, набранный пропорциональным шрифтом, легче читать, чем текст, набранный шрифтом с постоянным шагом, и, кроме того, он лучше смотрится.

Система Windows использует системный шрифт для меню, заголовков окон, элементов управления блоками диалога и, конечно, в качестве шрифта по умолчанию в любом контексте устройства, с которым связана программа. Фирма Microsoft объявила о том, что каждый драйвер дисплея Windows должен обеспечивать системный шрифт, который позволяет выводить не менее 25 строк и 80 столбцов текста. Это гарантирует, что программы для Windows смогут воспроизвести, по крайней мере, столько же текста, сколько и программы в MS DOS.

Для обеспечения совместимости с программами, написанными для ранних версий Windows (версии 1.x и 2.x), каждый драйвер дисплея также поддерживает системный шрифт с постоянным шагом. Если GDI обнаруживает, что программа была написана для старых версий Windows, он обеспечивает тот системный шрифт, который ожидает эта программа.

В табл.10.1 перечислены характеристики шрифта по умолчанию для четырех известных дисплеев и четырех известных принтеров.

Таблица 10.1 Шрифты по умолчанию.

Устройство	Размер поверхности рисования	Шрифт по умолчанию	
		Высота	Ширина
Дисплей CGA	640 x 200	8	7
— " — EGA	640 x 350	12	7
— " — VGA	640 x 480	16	7
— " — 8514/a	1024 x 768	20	9
Принтер Apple Imagewriter II	2550 x 3300	42	25
— " — Epson		25	36
LQ-1050(24 иглы)	3060 x 1980	50	30
Принтер HP LaserJet II	2550 x 3300	10	12
— " — Okidata ML 320 (9 игл)	1020 x 792		

Как видно из таблицы, размеры шрифта по умолчанию в разных устройствах сильно отличаются. Чтобы текст, созданный Вашей программой, хорошо смотрелся на любом устройстве, необходимо запросить GDI о размере шрифта до начала вывода текста. Для этой цели GDI имеет две подпрограммы: GetTextExtent и GetTextMetrics.

Подпрограмма GetTextExtent

Подпрограмма GetTextExtent вычисляет размер строки текста, используя шрифт, помещенный в данное время в контексте устройства. Вот например, как получить ширину и высоту фразы "Независимый от устройства" (Device Independent):


```

DWORD dwSize;
WORD yHeight, xWidth;

dwSize = GetTextExtent (hDC, "Device Independent", 18);
yHeight = HIWORD (dwSize);
xWidth = LOWORD (dwSize);

```

Эта подпрограмма принимает три параметра: логический номер контекста устройства, длинный указатель на строку и количество символов. Значение возврата имеет тип DWORD (длинное целое без знака), в котором упакованы значения высоты и ширины текста. Макропроцедуры HIWORD и LOWORD выделяют эти значения.

Подпрограмма GetTextMetrics

Подпрограмма GetTextMetrics обеспечивает более полный набор измерений шрифта. Каждый физический шрифт имеет запись-заголовок, которая включает в себя информацию о метрике шрифта. Есть 20 полей, определенных в структуре данных TEXTMETRIC. Ниже приведен вызов подпрограммы, в котором ведется поиск метрики шрифта, помещенного в данное время в контексте устройства:

```

TEXTMETRIC tm;
GetTextMetrics (hDC, &tm);

```

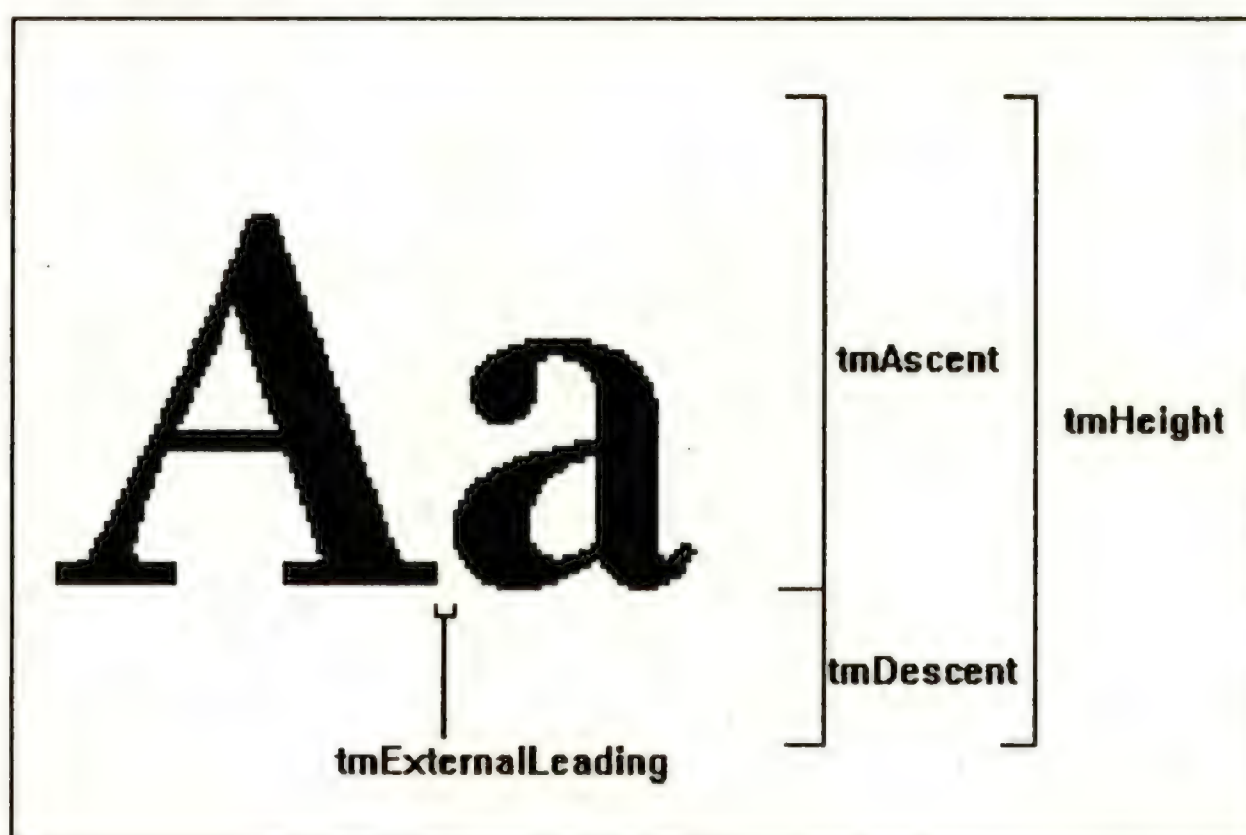


Рис.10.9. Определение ключевых полей TextMetric.

Поле tmHeight определяет размер символов шрифта в пикселях (если мы используем режим отображения MM_TEXT) или в элементах выбранного в данный момент режима отображения. Это поле имеет два компонента: tmAscent, который определяет высоту символа над нижней линией (размер строчной буквы вместе с верхним выносным элементом), и tmDescent, который определяет высоту символа под нижней линией для таких букв, как "g" и "y" (размер нижнего выносного элемента буквы).

Поле tmInternalLeading описывает размер области для таких диакритических знаков, как знаки ударения, умляути и т.п. Необходимо отметить, что в двух соседних строках текста верхняя граница диакритических знаков в нижней строке может соприкоснуться с границей нижнего выносного элемента верхней строки. Это не является отклонением для многих шрифтов.

Если же разработчик шрифта хочет избежать такой ситуации, то он может это сделать с помощью поля tmExternalLeading. Значение этого поля - предлагаемый интервал между строками. В описанной ситуации оно равно нулю. Термин "leading" (набор на шпоны) восходит к тем временам, когда шрифт вручную отливался в шпонах. Для разделения строк текста наборщики должны были вставлять тонкие полосы шпонов между рядами набора.

Вы могли заметить, что во многих примерах программ для высоты строки одновременно используются `tmHeight` и `tmExternalLeading`, как например:

```
int yLineHeight;  
TEXTMETRIC tm;  
GetTextMetrics (hDC, &tm);  
yLineHeight = tm.tmHeight + tm.tmExternalLeading;
```

Это общепринятый способ для задания интервалов между строками.

Попытаемся теперь изучить использование различных шрифтов и вывод многих строк текста.

Создание и использование логических шрифтов

Как мы обсуждали ранее, логический шрифт - это запрос. Он обеспечивает способ, с помощью которого программа описывает необходимый для использования физический шрифт. Логические шрифты создаются двумя подпрограммами: `CreateFont` и `CreateFontIndirect`. Подпрограммы выдают одни и те же результаты. Разница лишь в способе передачи параметров. В подпрограмме `CreateFont` 14 параметров. Подпрограмма `CreateFontIndirect` принимает всего один параметр - указатель на структуру, содержащую те же 14 значений. Мы собираемся ограничиться обсуждением подпрограммы `CreateFontIndirect`, так как ее несколько легче использовать.

Подпрограмма `CreateFontIndirect` принимает один параметр: указатель на структуру `LOGFONT`, которая определена в `WINDOWS.H` следующим образом:

```
typedef struct tagLOGFONT  
{  
    int    lfHeight;        /* Высота символа */  
    int    lfWidth;         /* Средняя ширина */  
    int    lfEscapement;     /* Угол наклона текста */  
    int    lfOrientation;   /* Угол наклона символа */  
    int    lfWeight;        /* Среднее число пикселей/1000 */  
    BYTE   lfItalic;        /* Flag != 0, если курсив */  
    BYTE   lfUnderline;     /* Flag != 0, если подчеркнуто */  
    BYTE   lfStrikeOut;     /* Flag != 0, если вычеркнуто */  
    BYTE   lfCharSet;       /* Набор символов: ANSI, OEM */  
        /* ANSI - Американский национальный институт стандартов */  
        /* OEM - (фирма-)изготовитель комплектного оборудования */  
    BYTE   lfOutPrecision;   /* Точность отображения - не используется */  
    BYTE   lfClipPrecision;  /* Точность вырезки - не используется */  
    BYTE   lfQuality;        /* Качество: черновое или пробное */  
    BYTE   lfPitchAndFamily; /* Флаги для стиля шрифта */  
    BYTE   lfFaceName[LF_FACESIZE]; /* Название шрифта */  
} LOGFONT;
```

В этой структуре совсем не много полей, но давайте посмотрим на поля, используемые в примере программы: `lfFaceName`, `lfHeight`, `lfWidth`, `lfItalic` и `lfUnderline`.

`lfFaceName` - это 32-символьное поле для названия шрифта. Программа может использовать названия шрифтов для того, чтобы информировать пользователей об имеющихся в наличии шрифтах. Ниже приведен перечень названий шрифтов для основных шрифтов Windows:

Courier
Helv
Modern
Roman
Script
Symbol
System
Terminal
Tms Rmn

Каждый шрифт имеет различные размеры и стили.

Поле `lfHeight` идентично полю `tmHeight` в структуре данных `TEXTMETRIC`. Поскольку мы рассматриваем только систему отображения `MM_TEXT`, то элементы - это пиксели. При использовании другой координатной системы `GDI` преобразует значения в ту систему отображения, которая в данный момент помещена в контексте устройства.

Поле `lfWidth` представляет собой среднюю ширину символов в шрифте. В структуре данных `TEXTMETRIC` также есть поле `tmAveCharWidth`, которое содержит идентичную информацию.

Поле `lfItalic` - это флаг. Ненулевое значение запрашивает курсив. Поле `lfUnderLine` - тоже флаг. Ненулевое значение запрашивает шрифт с подчеркиванием.

Наша следующая программа выводит текст с использованием трех различных шрифтов. Первый - это системный шрифт, который уже установлен для нас в контексте устройства. Два других, `Times Roman (Tms Rmn)` и `Helvetica (Helv)`, мы запрашиваем путем определения логического шрифта.

Программа выводит первые восемь полей `TEXTMETRIC` и название для каждого шрифта.

Three Fonts					
System		Tms Rmn		Helv	
Height	16	Height	16	Height	16
Ascent	13	Ascent	13	Ascent	13
Descent	3	Descent	3	Descent	3
Internal Leading	3	Internal Leading	3	Internal Leading	3
External Leading	0	External Leading	0	External Leading	0
Ave Char Width	7	Ave Char Width	6	Ave Char Width	5
Max Char Width	14	Max Char Width	14	Max Char Width	17
Weight	700	Weight	400	Weight	400

Рис.10.10. Примеры текста с использованием трех наборов шрифтов.

Ниже приведен текст нашей программы:

MAKEFILE.MAK

.AUTODEPEND

Определения транслятора

INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;
C:\BORLANDC\INCLUDE

CC = bcc -c -D_CLASSDLL -H -ml -WS -w -I\$(INC)

Неявные правила

.c.obj:
\$(CC) {S< }

.cpp.obj:
\$(CC) {S< }

Явные правила

TxtLines.exe: TxtLines.res TxtLines.def TxtLines.obj
tlink /c /C /n /P - /Twe /x @TxtLines.LNK
RC TxtLines.res TxtLines.exe

Зависимости отдельных файлов

TxtLines.obj: TxtLines.cpp

TxtLines.res: TxtLines.rc TxtLines.cur TxtLines.ico
RC -R -FO TxtLines.res TxtLines.RC

TXTLINES.LNK

c:\borlandc\lib\cOwl.obj+
TxtLines.obj
TxtLines, TxtLines
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwl.lib+
import.lib+
mathl.lib+
cl.lib
TxtLines.def

TXTLINES.CPP

/*

TXTLINES.CPP. Пример вывода текста с использованием трех наборов шрифтов

*/

#include <owl.h>

/*

Объявления классов

*/

class TTxtLinesApplication : public TApplication

{

public:

TTxtLinesApplication (LPSTR lpszName, HANDLE hInstance,
HANDLE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow);

virtual void InitMainWindow ();

};

class TTxtLinesWindow : public TWindow

{

public:

TTxtLinesWindow (PTWindowsObject pwParent,
LPSTR lpszTitle, PTModule pmModule);

-TTxtLinesWindow ();

virtual LPSTR GetClassName ();

virtual void GetWindowClass (WNDCLASS&);

virtual void Paint(HDC hdc, PAINTSTRUCT& ps);

private:

HANDLE hfontTmsRmn;

HANDLE hfontHelv;

};

/*

Прототипы функций

*/

void NEAR PASCAL TxtWriteTextMetrics (HDC hdc, int xText, int yText);

/*

Основная функция: WinMain

*/

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
LPSTR lpszCmdLine, int nCmdShow)

{

TTxtLinesApplication TxtLines ("TxtLines", hInstance, hPrevInstance,
lpszCmdLine, nCmdShow);

TxtLines.Run();


```

    return TxtLines.Status;
}

/*
Компонентная функция класса прикладной программы
*/
TtxtLinesApplication::TtxtLinesApplication
    (LPSTR lpszName,
     HANDLE hInstance,
     HANDLE hPrevInstance,
     LPSTR lpszCmdLine, int nCmdShow)
    :TApplication (lpszName, hInstance,
                  hPrevInstance, lpszCmdLine,
                  nCmdShow)
{
    /*Процедура инициализации прикладной программы */
}

/*
Компонентная функция класса прикладной программы
*/
void TtxtLinesApplication::InitMainWindow ()
{
    MainWindow = new TtxtLinesWindow(NULL, "Three Fonts", NULL);
}

/*
Компонентная функция класса TtxtLinesWindow
*/
TtxtLinesWindow::TtxtLinesWindow
    (PTWindowsObject pwParent,
     LPSTR lpszTitle, PTModule pmModule)
    :TWindow(pwParent, lpszTitle, pmModule)
{
    LOGFONT lf;

    memset (&lf, 0, sizeof (LOGFONT));
    lf.lfHeight = 16;
    lf.lfWidth = 6;
    lf.lfUnderline = 1;
    lstrcpy ((LPSTR)&lf.lfFaceName[0], (LPSTR)"Tms Rmn");
    hfontTmsRmn = CreateFontIndirect (&lf);

    memset (&lf, 0, sizeof (LOGFONT));
    lf.lfHeight = 16;
    lf.lfWidth = 7;
    lf.lfUnderline = 1;
    lstrcpy ((LPSTR)&lf.lfFaceName[0], (LPSTR)"Helv");
    hfontHelv = CreateFontIndirect (&lf);
}

/*
Компонентная функция класса TtxtLinesWindow
*/
TtxtLinesWindow::~TtxtLinesWindow()
{
    DeleteObject (hfontTmsRmn);
    DeleteObject (hfontHelv);
}

```



```

}

/*
Компонентная функция класса TTxtLinesWindow
*/
LPSTR TTxtLinesWindow::GetClassName ()
{
    return "TxtLines:MAIN";
}

/*
Компонентная функция класса TTxtLinesWindow
*/
void TTxtLinesWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
}

/*
Компонентная функция класса TTxtLinesWindow
*/
void TTxtLinesWindow::Paint(HDC hdc, PAINTSTRUCT& ps)
{
    DWORD dwSize;
    int xText;
    int yText;
    RECT r;

    dwSize = GetTextExtent (hdc, "X", 1);
    GetClientRect (HWindow, &r);
    xText = LOWORD (dwSize) * 2;
    yText = 10;
    TxtWriteTextMetrics (hdc, xText, yText);
    SelectObject (hdc, hfontTmsRmn);
    xText += r.right/3;
    TxtWriteTextMetrics (hdc, xText, yText);
    SelectObject (hdc, hfontHelv);
    xText += r.right/3;
    TxtWriteTextMetrics (hdc, xText, yText);
}

/*
TxtWriteTextMetrics - Запись метрик текущего набора шрифтов
*/
void NEAR PASCAL TxtWriteTextMetrics (HDC hdc, int xText, int yText)
{
    char          buffer[80];
    DWORD         dwSize;
    int           nLength;
    int           yLineHeight;
    int           xIndent;
    TEXTMETRIC    tm;
    static char    *apchLabel[] =
    {
        "Height",          /* [0] */
        "Ascent",          /* [1] */
    }

```



```

    "Descent",          /* [2] */
    "Internal Leading", /* [3] */
    "External Leading", /* [4] */
    "Ave Char Width",   /* [5] */
    "Max Char Width",   /* [6] */
    "Weight"            /* [7] */
};

```

```

GetTextMetrics (hdc, &tm);
yLineHeight = tm.tmHeight + tm.tmExternalLeading;
nLength = GetTextFace (hdc, sizeof(buffer)-1, buffer);

TextOut (hdc, xText, yText, buffer, nLength);
yText += yLineHeight * 2;
dwSize = GetTextExtent (hdc, apchLabel[4], lstrlen (apchLabel[4]));
xIndent = LOWORD (dwSize);
dwSize = GetTextExtent (hdc, "xxxx", 4);
xIndent += LOWORD (dwSize);

TextOut (hdc, xText, yText, apchLabel[0], lstrlen (apchLabel[0]));
nLength = wsprintf (buffer, "%d", tm.tmHeight);
SetTextAlign (hdc, TA_RIGHT);
TextOut (hdc, xText + xIndent, yText, buffer, nLength);
SetTextAlign (hdc, TA_LEFT);
yText += yLineHeight;

TextOut (hdc, xText, yText, apchLabel[1], lstrlen (apchLabel[1]));
nLength = wsprintf (buffer, "%d", tm.tmAscent);
SetTextAlign (hdc, TA_RIGHT);
TextOut (hdc, xText + xIndent, yText, buffer, nLength);
SetTextAlign (hdc, TA_LEFT);
yText += yLineHeight;

TextOut (hdc, xText, yText, apchLabel[2], lstrlen (apchLabel[2]));
nLength = wsprintf (buffer, "%d", tm.tmDescent);
SetTextAlign (hdc, TA_RIGHT);
TextOut (hdc, xText + xIndent, yText, buffer, nLength);
SetTextAlign (hdc, TA_LEFT);
yText += yLineHeight;

TextOut (hdc, xText, yText, apchLabel[3], lstrlen (apchLabel[3]));
nLength = wsprintf (buffer, "%d", tm.tmInternalLeading);
SetTextAlign (hdc, TA_RIGHT);
TextOut (hdc, xText + xIndent, yText, buffer, nLength);
SetTextAlign (hdc, TA_LEFT);
yText += yLineHeight;

TextOut (hdc, xText, yText, apchLabel[4], lstrlen (apchLabel[4]));
nLength = wsprintf (buffer, "%d", tm.tmExternalLeading);
SetTextAlign (hdc, TA_RIGHT);
TextOut (hdc, xText + xIndent, yText, buffer, nLength);
SetTextAlign (hdc, TA_LEFT);
yText += yLineHeight;

TextOut (hdc, xText, yText, apchLabel[5], lstrlen (apchLabel[5]));
nLength = wsprintf (buffer, "%d", tm.tmAveCharWidth);
SetTextAlign (hdc, TA_RIGHT);
TextOut (hdc, xText + xIndent, yText, buffer, nLength);
SetTextAlign (hdc, TA_LEFT);

```



```

yText += yLineHeight;
TextOut (hdc, xText, yText, apchLabel[6], strlen (apchLabel[6]));
nLength = wsprintf (buffer, "%d", tm.tmMaxCharWidth);
SetTextAlign (hdc, TA_RIGHT);
TextOut (hdc, xText + xIndent, yText, buffer, nLength);
SetTextAlign (hdc, TA_LEFT);
yText += yLineHeight;

TextOut (hdc, xText, yText, apchLabel[7], strlen (apchLabel[7]));
nLength = wsprintf (buffer, "%d", tm.tmWeight);
SetTextAlign (hdc, TA_RIGHT);
TextOut (hdc, xText + xIndent, yText, buffer, nLength);
SetTextAlign (hdc, TA_LEFT);
yText += yLineHeight;
}

```

TXTLINES.RC

```

snapshot icon TxtLines.ico
hand cursor TxtLines.cur

```

TXTLINES.DEF

```

NAME TXTLINES
EXETYPE WINDOWS
DESCRIPTION 'TxtLines - Sample text output'
CODE MOVEABLE DISCARDABLE
DATA MOVEABLE MULTIPLE

HEAPSIZE 512
STACKSIZE 5120

```

Для создания логического шрифта эта программа устанавливает желаемые значения полей в структуре LOGFONT и вызывает CreateFontIndirect. Подпрограмма игнорирует нулевые значения и создает запрос на шрифт, используя явно установленные поля, поэтому для обнуления всех полей мы вызываем memset из библиотеки C.

Подобно другим объектам GDI, логические шрифты занимают оперативную память, которая должна быть освобождена для дальнейшего использования. В ответ на сообщение WM_DESTROY программа удаляет логический шрифт с помощью вызова DeleteObject. Основная часть работы в нашей программе выполняется в ответ на сообщение WM_PAINT. Вначале с помощью GetClientRect программа получает ширину окна и разбивает его на три столбца. Сначала мы выводим информацию о метрике для системного шрифта, а затем переходим к шрифтам Times Roman и Helvetica. Необходимо отметить, что, хотя логический шрифт уже был создан, физический шрифт не будет определен до тех пор, пока логический номер логического шрифта не будет помещен в контексте устройства:

```
SelectObject (hDC, hfontTmsRmn);
```

Наша подпрограмма TxtWriteTextMetrics выводит первые восемь полей структуры данных TEXTMETRIC для каждого из трех шрифтов. Для того чтобы получить название каждого шрифта, мы вызываем подпрограмму GetTextFace, которая определяется следующим образом:

```
GetTextFace (hDC, nCount, lpFaceName);
```


где `hDC` - логический номер, помещенный в контекст устройства; `nCount` - размер буфера для получения названия шрифта; `lpFaceName` - длинный указатель на буфер символов.

Мы вызвали `GetTextExtent` дважды: сначала для того, чтобы определить длину самого длинного имени поля "External Leading", а затем - чтобы определить длину "XXXX" (представление числового значения). Переменная `xIndex` хранит сумму этих двух длин и используется в качестве границы между именами полей и значениями.

Обратите внимание на то, что для каждого поля `TEXTMETRIC` мы вывели имя поля, а затем вызвали подпрограмму `wsprintf` для преобразования числового значения в текстовую строку, которая может быть выведена подпрограммой `TextOut`. Подпрограмма `wsprintf` аналогична подпрограмме `sprintf` из библиотеки C, но она свободна от ряда несовместимостей `sprintf` в среде Windows.

Заметьте также, что мы использовали атрибут выравнивания текста (через `SetTextAlign`) для создания таблицы, в которой текст выровнен слева, а числовые значения - справа.

Как Вы видите, GDI рассматривает текст как графический объект. Это означает, что Вы владеете множеством элементов управления размещением текста, изменением размеров и установкой цветов текста. Вы можете легко объединять текст и геометрические фигуры, соединять на одной странице текст различных размеров и стилей. Мы считаем, что эти дополнительные усилия вполне оправдываются возможностью не зависимость от устройства набивки текста.

Тем программистам, которым не требуется большое разнообразие в тексте, GDI гарантирует шрифт по умолчанию для каждого устройства. Это обеспечит хорошее представление выдаваемого текста при минимальных усилиях с Вашей стороны.

Часть 4. Объекты пользовательского интерфейса

Глава 11. Команды меню и кодовые комбинации клавиш

В среде Windows для вызова нужной команды существует два вида объектов пользовательского интерфейса - меню и кодовые комбинации клавиш. Кодовые комбинации значительно ускоряют работу пользователя, а меню облегчает изучение данного программного продукта. С его помощью пользователь может выбрать нужное действие или опцию. Для того чтобы пользователю было легче перейти от работы с меню к работе с кодовыми комбинациями клавиш, они приведены в нем рядом с названиями команд. На рис.11.1 представлен стандартный вид меню, где поясняется, какому объекту соответствует та или иная кодовая комбинация клавиш. Например, согласно рисунку, нажав клавиши Shift и Del, пользователь может без меню вызвать команду Cut.

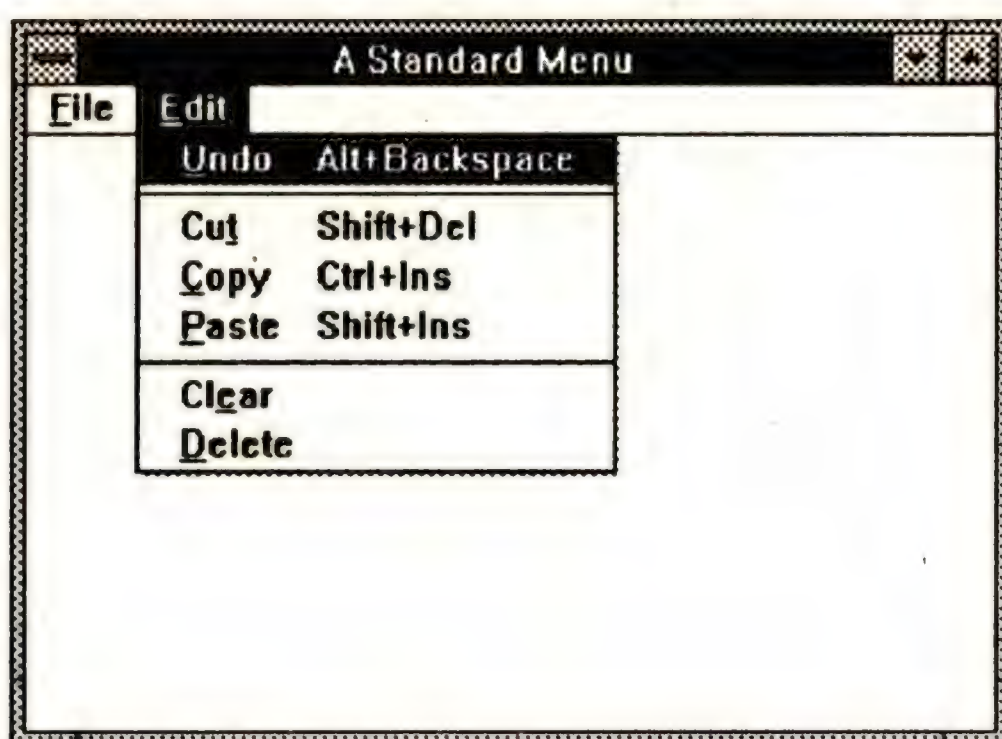


Рис.11.1. Вид меню с наименованиями кодовых комбинаций клавиш.

Как показывает опыт, программы и программные среды, созданные на основе стандартов пользовательского интерфейса среды Windows, осваиваются пользователями легче, чем все остальные. Поэтому программистам, которые собираются создавать пользовательский интерфейс с меню и кодовыми комбинациями клавиш, следует хорошенько изучить все эти стандарты. Сделать это можно двумя способами: первый - изучать эти стандарты на практике, в процессе работы со средой Windows, а второй - прочитать руководство, которое входит в состав документации, поставляемой фирмой Microsoft вместе с программным продуктом SDK. Лучше всего, конечно, изучать эти стандарты, читая руководство, это позволяет пользователю сэкономить время при изучении возможностей программной среды. Пользователи, которые предпочитают изучать возможности программной среды на практике, зачастую сталкиваются с большими трудностями из-за того, что программный продукт в той или иной мере не соответствует общепринятым нормам.

Давайте кратко остановимся на том, как пользоваться меню.

Стандарты пользовательского интерфейса

При изучении меню пользователю приходится иметь дело с двумя типами объектов: командами и опциями. Команды в меню - это слово (чаще всего, глагол) или словосочетание (глагол или существительное), обозначающее определенное действие над объектом, указанным пользователем. Например, во многих программах в меню File часто встречается действие Open, открывающее нужный файл. Команды влияют

на характеристики всего программного продукта в целом, например, в системном меню, которое открывается в верхней части основного окна редактора, содержится команда Close, закрывающая окно редактора и приводящая к завершению работы всей программы. Управляющие параметры командного языка среды могут изменять свои значения на противоположные, а в командах меню, которые используются временно, параметры задают режимы работы программной среды на длительный срок до тех пор, пока их не изменят. Например, в оболочке Program Manager есть параметр, позволяющий вводить и отменять два режима: Auto Arrange и Minimize on Use.

Часто в программном продукте предусматривают специальное окно, где указывают, какие из опций активизированы, а какие - нет. Наиболее важные параметры в сложных программах обычно выносят в меню, а остальные, для того чтобы пользователь мог их изменять, размещают в специальных дополнительных окнах, которые называют диалоговыми. Другими словами, все параметры в основное меню выносить не надо. Те из них, которыми пользуются редко, следует размещать в диалоговых окнах. О том, что они собой представляют, мы поговорим в главе 14.

Порой вслед за вызовом того или иного объекта из меню начинается диалог между пользователем и машиной, который ведется с помощью упомянутых выше диалоговых окон. В этом случае для того чтобы указать, какие особенности имеет тот или иной объект, входящий в меню, рядом с его названием проставляют так называемые метки, которые следуют за названием объекта, например многоточие (...). Приведем некоторые из таких меток.

1. Кодовые комбинации клавиш сообщают пользователю, что вместо вызова меню можно вызвать команду с помощью заданной кодовой комбинации клавиш.
2. Стрелка означает, что, когда пользователь с помощью мыши или клавиши перемещения курсора выберет данный объект, появится всплывающее меню.
3. Разделительная черта отделяет одну группу объектов меню от другой.
4. Галочка означает, что выбран данный параметр меню.
5. Многоточие, стоящее после названия объекта меню, сообщает пользователю о том, что при выборе данного объекта появится диалоговое окно.
6. Восклицательный знак, стоящий после имени объекта, означает, что данный объект сразу вызывает действие, меню при этом на экране не появляется.
7. Полутоновая надпись означает, что в текущем состоянии программной среды данный объект отсутствует. В системном меню, когда окно, например, максимально расширено, некоторые надписи превращаются в полутоновые.
8. Знак подчеркивания под буквой названия объекта означает, что данный объект можно вызвать с помощью кодовой комбинации, в которую входит эта буква. Такие буквы называют мнемоническими. Например, для того чтобы вызвать то или иное меню, необходимо нажать клавишу Alt и клавишу с буквой, которая в названии данного меню была выделена подчеркиванием, а затем, когда меню появилось, выбрать в нем нужную команду. На рис.11.2 приведен пример меню с различными метками.

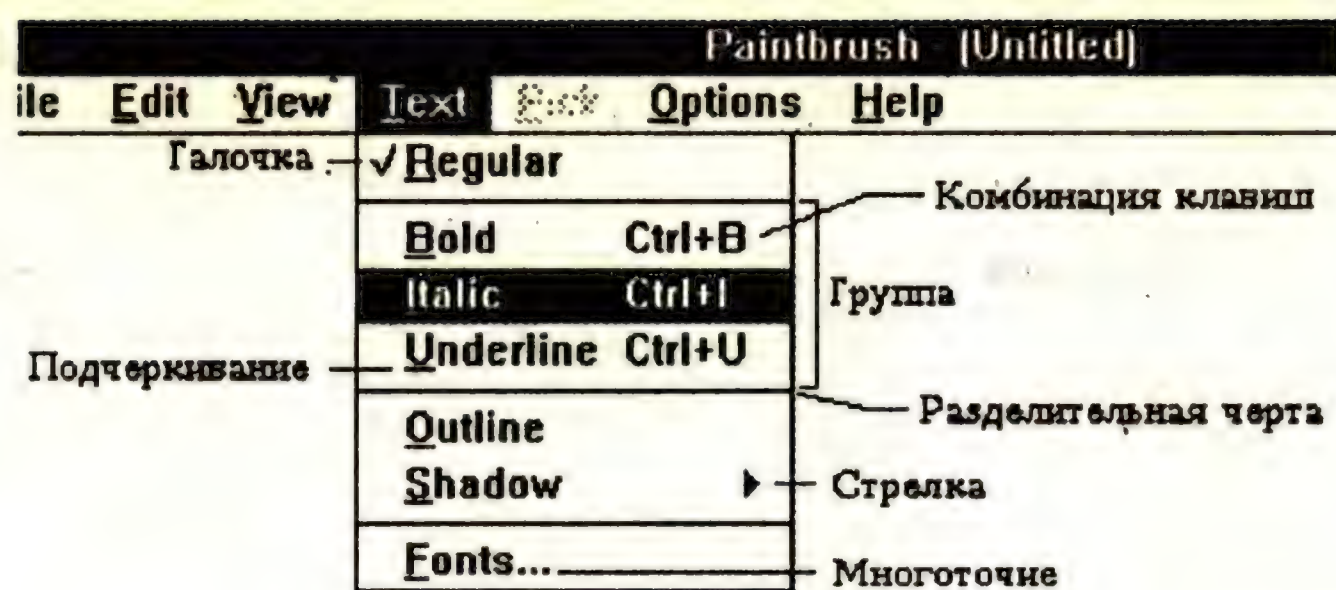


Рис.11.2. Пример меню с различными специальными обозначениями.

Для удобства пользователя в программных средах, как правило, создают системное меню. С точки зрения программирования создать его несложно. Без особой нужды содержимое меню изменять не следует. Как правило, туда можно лишь кое-что добавить. На рис.11.3 приведен стандартный вид системного меню.

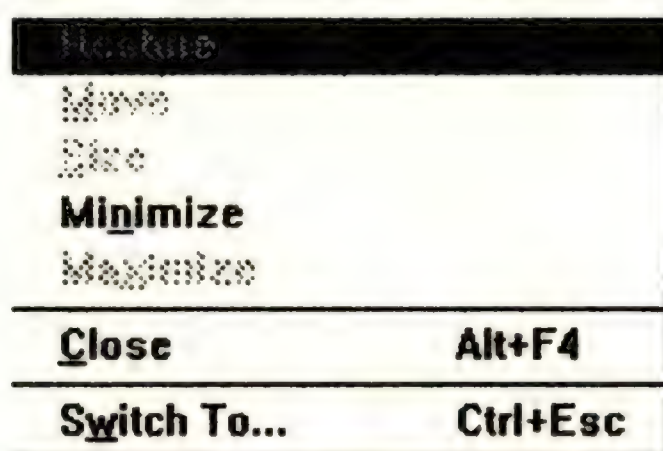


Рис.11.3. Стандартный вид меню System.

В тех программных оболочках, где пользователям приходится работать с файлами, для удобства следует предусмотреть меню File, обеспечивающее пользователю доступ к командам, которые связаны с открыванием, закрыванием и распечатыванием файлов. Заметим попутно, что обычно в меню File имеется команда под названием Exit, которая вызывает завершение работы программной оболочки. Фактически эта команда дублирует команду Close, которая содержится в системном меню (рис.11.4), но для удобства пользователей ее все же следует продублировать. Если Вы создаете программную оболочку, которая не будет содержать меню File, то следует команду Exit поместить в нижней части первого из тех меню, которые будут вызываться при работе с этой командой.



Рис.11.4. Стандартный вид меню File.

Для удобства пользователей Вашей программной оболочки следует также создать меню Edit. Это меню содержит список операций редактирования текстов, включая операции Clipboard, поиск фрагмента текста и его замену, повторный вызов и выполнение предыдущих команд и т.п. В разрабатываемой программе можно дополнительно предусмотреть команды, которые позволяют обращаться с ее компонентами так, как Вы этого пожелаете. Например, в программе для обработки текстов из меню

Edit можно предусмотреть действия, которые позволяют изменять форму заголовка документа, сокращать или растягивать его и т.п. На рис.11.5 мы приводим стандартный вид меню Edit. Далее приведем пример, где покажем, как создавать такие меню.

Edit	
U ndo	Alt+Backspace
C ut	Shift+Del
C opy	Ctrl+Ins
P aste	Shift+Ins
C lear	
D elete	

Рис.11.5. Стандартный вид меню Edit.

Доступ к содержимому меню осуществляют как перемещением мыши, так и с помощью клавиатуры. Например, для того чтобы вызвать меню на экран, подводят указатель мыши к нужной надписи и дважды нажимают клавишу, а затем, чтобы выбрать нужную команду или опцию, осуществляют ввод с клавиатуры или, наоборот, для того чтобы вызвать меню, сначала набирают соответствующую кодовую комбинацию клавиш в сочетании с Alt или одну из функциональных клавиш, например F10, а затем, находясь в меню, выбирают нужную команду, управляя мышью. Некоторые разработчики ориентируются исключительно на клавиатуру. Так бывает либо потому, что они привыкли работать с ней, либо у них отсутствует мышь, но в основном в программах стараются предусмотреть возможность их дублирования. Разумеется, добиться их полной взаимозаменяемости нельзя, потому что хотя бы, что с помощью мыши трудно вводить символьные данные. Но пользоваться и клавиатурой, и мышью при работе с меню - неплохая идея, и о ней не следует забывать.

Более подробно ввод команд и управление меню как с помощью мыши, так и клавиатуры, мы рассмотрим в главах 15 и 16. Там мы вновь вернемся к этой идее, которая является важной частью общего замысла программной среды Windows. А сейчас остановимся на тонкостях программирования при создании меню.

Вопросы программирования при создании меню

В среде Windows создана гибкая и в то же время понятная всем система меню, поэтому с ней удобно работать. Именно создание этой системы при разработке среды Windows было наиболее сложной и интересной частью всей работы над ней, потому что другая часть - создание самих меню - для программистов сложности не представляла. Над созданием системы меню бились долго, и не раз коды ее поддержки перерабатывали, перенастраивали и переписывали заново.

Главная трудность заключалась в том, что эти меню должны были появляться и исчезать в один миг, чтобы диалог пользователя был живым, занимал меньше времени. С этой целью с того места экрана, где должно было появиться меню, снимали как бы моментальную фотографию - битовый образ, который вырезали, а на его место помещали меню, когда работа с ним заканчивалась, меню убирали, а битовый образ помещали вновь на то же самое место.

Создать меню не столь уж сложно. Фирма Microsoft сделала многое для того, чтобы эта работа не утомляла и занимала меньше времени. Сейчас мы с Вами увидим, что меньше всего времени уйдет, если мы будем создавать меню, заносая краткие сведения о нем в резервный файл Вашей программы. Когда в него вносят сведения о классе окна просмотра, один из его параметров - WNDCLASS - принимает значение имени меню. Теперь, если окно этого класса создано, появится и соответствующее меню. Программисту остается лишь обработать сообщение типа WM_COMMAND, которое это меню передаст оконной процедуре его программы. Меню, созданное в среде Windows, где общение с пользователем осуществляется с помощью сообщений, будет работать с Вашей программой также с помощью сообщений, вернее не с ней, а с ее так называемой оконной процедурой. Наиболее важным сообщением, которое

передает меню, является сообщение WM_COMMAND, означающее, что тот или иной параметр меню определен.

Названия различных этапов работы меню, возникающие сообщения и соответствующие им операции с мышью и клавиатурой приведены в табл.11.1.

Таблица 11.1. Стадии работы меню и соответствующие сообщения.

Этап работы	Операции с мышью	Операции с клавиатурой	Сообщение
Инициализация меню		F10 или клавиша Alt	WM_INITMENU
Стадия	Операции с мышью	Операции с клавиатурой	Сообщение
Вывод всплывающего меню на экран		Клавиши курсора или мнемонические	WM_INITMENUPOPUP
Инициализация и вывод на экран всплывающего меню	Нажать клавишу	Alt+мнемоническая клавиша	WM_INITMENU и WM_INITMENUPOPUP
Нахождение нужного пункта в меню	Перемещение	Работа клавишами курсора	WM_MENUSELECT
Выбор нужного пункта в меню	Отпустить клавишу	Нажать клавишу Enter или мнемоническую клавишу	WM_COMMAND

Заметим, что вид сообщений, возникающих при диалоге пользователя с программой, не зависит от того, с чем он работает - с мышью или с клавиатурой. Эти сообщения большей частью содержатся в системе управления меню среды Windows, и львиную часть работы она берет на себя. Вашей программе надо как можно точнее следовать указаниям, содержащимся в ее сообщениях.

Теперь, очевидно, Вы хотите узнать, какова роль других сообщений, кроме WM_COMMAND, ведь порой нужны и они.

Если вдруг программа захочет инициализировать основное меню до того как пользователь закончит работу с ним, то она должна сперва дать отклик на сообщение WM_INITMENU, а прежде чем появится всплывающее меню, ей следует дать отклик на сообщение WM_INITMENUPOPUP. Отклики на эти инициализирующие сообщения часто используют в тех программах, где применяется Clipboard, для того, чтобы пользователь мог узнать, например, о том, есть ли в Clipboard какая-нибудь информация, которую можно было бы включить в окно. Если таковая есть, то надпись Paste в меню Edit выделяют более ярким цветом, если нет, ее изображают серыми полутонами.

В сообщении WM_MENUSELECT содержится информация для программы о том, на каком именно пункте меню остановился в данный момент пользователь в процессе поиска. Ее можно использовать при создании так называемых окон дополнительной информации (information area) для того, чтобы в них при необходимости сообщать нужные подробности о каждом пункте, входящем в меню. Такое в коммерческих программах среды Windows предусматривают, правда, очень редко. Как создавать окна и как работать с сообщениями WM_MENUSELECT, мы покажем на примере программы в главе 13, там, где будем обсуждать вопросы, связанные с созданием окон.

Шаблоны для меню

Простейшие меню начинают создавать с шаблона, который определяет, какие всплывающие меню и какие пункты будут в него входить.

Структура построения шаблона напоминает иерархическую структуру файловой системы DOS с ее корневыми каталогами и подкаталогами. Вершиной этой иерархической структуры, корневым каталогом, является строка окна, в которой перечислены пункты меню, горизонтальное меню. На этом верхнем уровне содержатся либо названия пунктов меню, которые формируют командные сообщения, либо заголовки всплывающих меню, которые появляются непосредственно под этими заголовками. Заметим, что такие всплывающие меню называют выпадающими.

Второй уровень иерархии шаблона, соответствующий уровню подкаталогов в файловой структуре DOS, занимают вложенные меню. Как и подкаталоги, которые содержат свои подкаталоги, вложенные меню также содержат свои вложенные меню. Число их ничем не ограничено, но здравый смысл подсказывает, что больше трех уровней меню делать не следует. Достаточно создать верхнюю линейку и вложенные меню первого и второго уровня. Больше - не надо. В противном случае пользователь, для которого Вы создаете программу, утонет в море меню.

Проще и легче всего создавать шаблоны для меню с помощью программы Resource Workshop. Эта программа облегчает (рис.11.6) внесение изменений в шаблон, и Вы можете сразу видеть результаты Ваших действий. Шаблон можно создавать и с помощью текстового редактора. Но какой бы метод Вы ни выбрали, наши рассуждения помогут Вам увидеть процесс создания шаблона во всех подробностях.

```
Menu ID MENU[load option][memory option]
```

```
Begin
```

```
Menuitem or POPUP statement
```

```
Menuitem or POPUP statement
```

```
Menuitem or POPUP statement
```

```
end
```

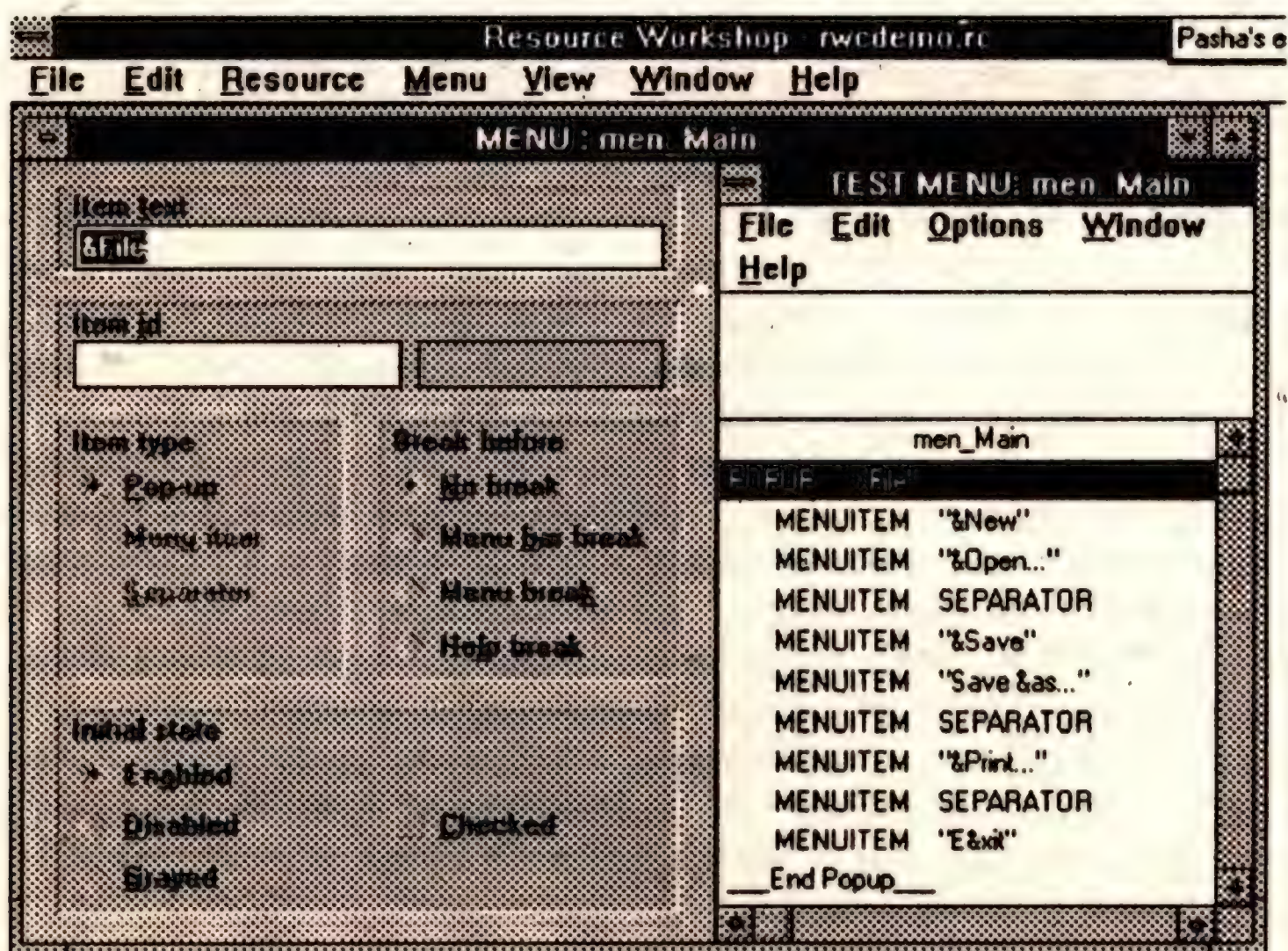


Рис.11.6. Разработка шаблона меню с помощью программы Resource Workshop.

В приведенном выше фрагменте программы параметр [load option] принимает одно из двух значений: PRELOAD или LOADONCALL, которые определяют порядок обращения к информации меню, как к объекту памяти. PRELOAD указывает, что ресурс меню будет загружен в память до того, как программа стартует, LOADONCALL - что каждый пункт меню будет загружен в нее только при необходимости; параметр [memory option] может принимать значения FIXED, MOVEABLE или ISCARDABLE, которые указывают на характер поведения объекта при его размещении в памяти. Более подробно этот параметр мы рассмотрим в главе 17. А сейчас давайте зададим значения этих двух параметров: LOADONCALL и DISCARDABLE. Это позволит нам показать, как создают меню.

Каждое выражение типа MENUITEM определяет пункт меню, при выборе которого формируется сообщение типа WM_COMMAND, а выражение типа POPUP предшествует определению всплывающего меню и отделяется от предыдущих и последующих выражений типа MENUITEM или POPUP выражениями BEGIN и END. Кстати, Вы можете немного сократить работу по набивке программы, если вместо Begin и END воспользуетесь их заменителями в языке C - фигурными скобками {, }. Приведем пример того, как можно создать меню File и Edit согласно рекомендациям, приведенным в этой главе:

7 MENU

```
{
  POPUP "& FILE"
  {
    MENUITEM "&NEW" 1
    MENUITEM "&OPEN" 2
    MENUITEM "&SAVE" 3
    MENUITEM "SAVE& AS" 4
    MENUITEM SEPARATOR
    MENUITEM "& Print" 5
    MENUITEM SEPARATOR
    MENUITEM "E&XIT" 6
  }
  POPUP "& EDIT"
  {
    MENUITEM "&UNDO\+Alt+Backspace" 7
    MENUITEM SEPARATOR
    MENUITEM "CU&T\+Shift+Del" 8
    MENUITEM "&COPY\+Ctrl+Ins" 9
    MENUITEM "&PASTE\+Shift+Ins" 10
    MENUITEM SEPARATOR
    MENUITEM "CL&EAR" 11
    MENUITEM "&DELETE" 12
  }
}
```

Идентификатором меню служит цифра 7. Можно преобразовать ее к виду, более удобному для чтения машиной, например в код ASCII. Идентификатор для меню служит именем в среде Windows, которое позволяет объявить и идентифицировать его. Знак & обозначает мнемоническую, т.е. соответствующую данной букве клавишу, которая используется для введения данного меню. Знак \t генерирует символ табуляции и служит также для того, чтобы отделить комбинацию кодовых клавиш от названия пункта в меню.

Одна из наиболее важных величин в описании любого меню, пожалуй, итоговый код команды - число, которое располагается вслед за каждым выражением MENUITEM и отделяет один пункт меню от другого. В системном меню этот код служит для того, чтобы отличать пункты меню от сообщений WM_COMMAND и WM_MENUSELECT. В общем случае выражение POPUP выглядит таким образом:

POPUP text[, option list]

А выражение MENUITEM

MENUITEM text, result code [, option list]
(текст) (итоговый код) (список опций)

Очевидно, что выражение MENUITEM отличается от POPUP тем, что включает итоговый код. В списке опций [options list] определяют пять параметров. Первые три задают исходное состояние пункта меню:

* CHECKED - устанавливает проверочный символ вслед за именем выпадающего меню или именем пункта меню. Проверяются пункты, содержащиеся в выпадающем меню, а в основном меню - не проверяются.

* GRAYED - в исходном состоянии пункт меню выключен и изображается серым полутонem.

* INACTIVE - пункт меню изображается обычным шрифтом, но его нельзя активизировать в обычном режиме. Параметр GRAYED позволяет пользователю видеть, можно воспользоваться данным пунктом меню или нет.

Оставшиеся два параметра определяют конфигурацию меню:

* MENUBREAK - вызывает разделение меню. Те меню, где символы размещаются по горизонтали (например, в горизонтальном меню), разделяют вертикальными линиями, а те меню, где символы или текст размещаются по вертикали, - горизонтальными линиями. Если надпись в меню не разделять, то может случиться, что все надписи в верхней линейке расположатся одна под другой, а в выпадающих и вложенных меню - друг за другом, и пользоваться ими будет невозможно.

* MENUBARBREAK - вызывает аналогичное разделение, но во всплывающем меню дополнительно появляется вертикальная разделительная черта.

В линейке обработки последних два параметра действуют одинаково и позволяют записать следующий пункт меню с новой строки. Рассмотрим следующий пример:

```
MENU
{
  MENUITEM      "Item-1", 1
  MENUITEM      "Item-2", 2, menubreak
  MENUITEM      "Item-3", 3,
  MENUITEM      "Item-4", 4,
  MENUITEM      "Item-5", 5, menubarbreak
  MENUITEM      "Item-6", 6
}
```

Как видно на рис.11.7, разделительные выражения MENUBREAK и MENUBARBREAK вызывают в тексте меню перескоки с одной строки на другую. Так, пункты 2 и 5 записаны с новой строки. Вообще-то среда Windows позволяет осуществлять перескоки текста всегда, когда строка текста не помещается по ширине в окне. Но, если Вы хотите сами задавать место и порядок перехода на новую строку, то без этих опций Вам не обойтись.

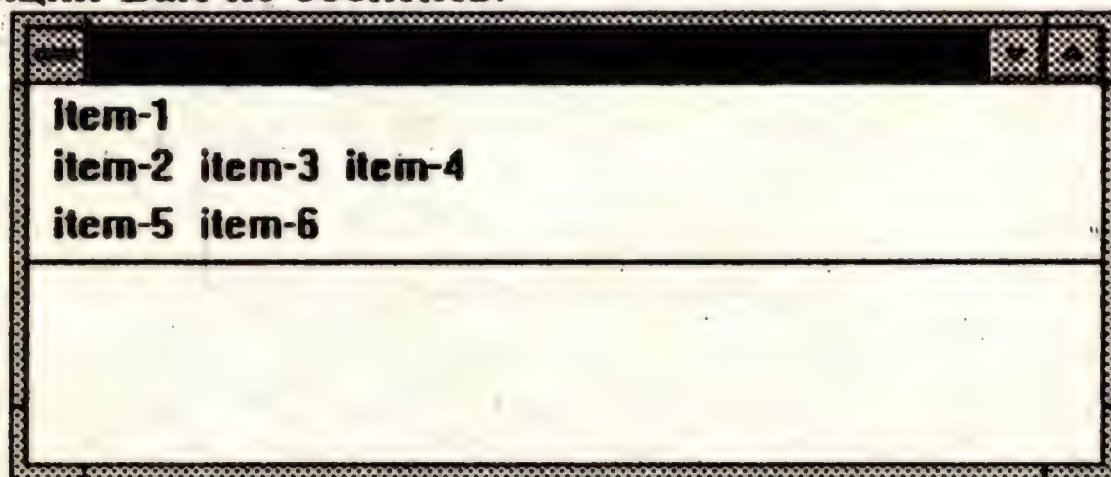


Рис.11.7. Результат работы функций MENUBREAK и MENUBARBREAK меню верхнего уровня.

Рассмотрим теперь, как работают эти параметры во всплывающем меню. В качестве примера приведем следующий фрагмент программы:

```
7 MENU
{
  POPUP"Popur"
  {
    MENUITEM "Item -1", 1
    MENUITEM "Item -2", 2, menubreak
    MENUITEM "Item -3", 3,
    MENUITEM "Item -4", 4,
    MENUITEM "Item -5", 5, menubarbreak
    MENUITEM "Item -6", 6
  }
}
```

На рис.11.8 приведено полученное меню. Заметим, что и тот, и другой параметр задают начало следующей записи в новой колонке, но опция **MENUBARBREAK** формирует, кроме того, еще и вертикальную черту, которая в создаваемом меню позволяет получившиеся колонки из названий отделить друг от друга.

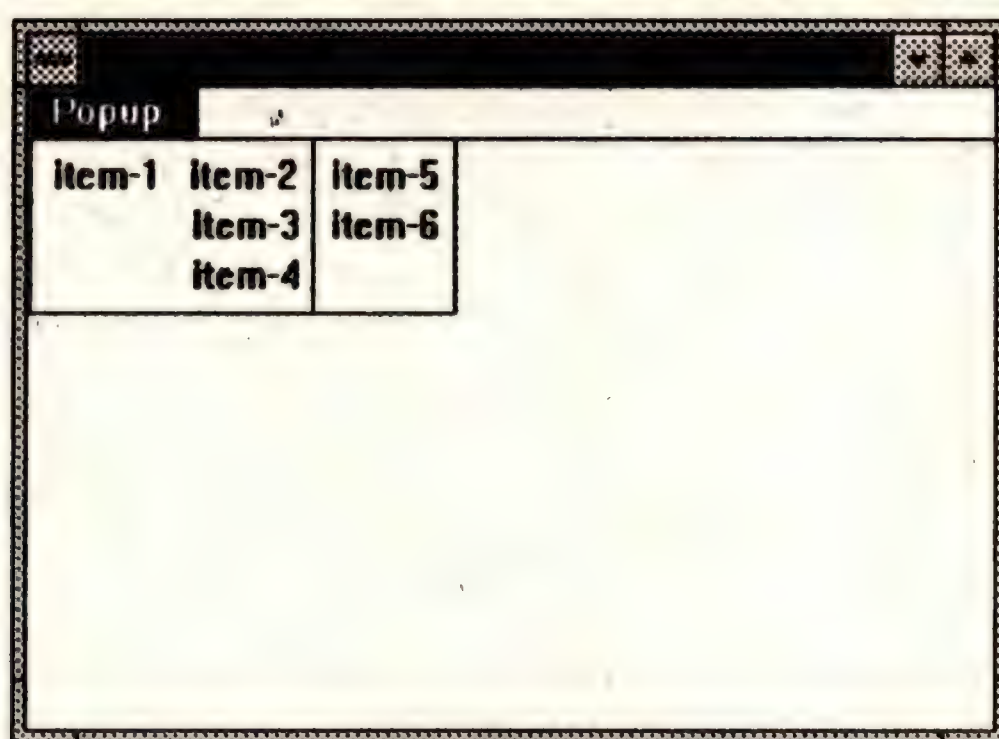


*Рис.11.8. Пример использования опций **MENUBREAK** и **MENUBARBREAK** во всплывающем меню.*

Помимо перечисленных выше, в описании языка упоминается еще один параметр **HELP**. В прежнем стандарте для **HELP** меню отводили место в правом верхнем углу экрана, правее всех остальных меню, и вертикальной чертой отделяли от них, однако в новом стандарте построения меню его нет, он оказывается ненужным. Попробуйте включить эту опцию в свою программу и Вы увидите, что системное меню никак не реагирует на ее появление. Но, с другой стороны, сейчас, если Вы хотите, чтобы всплывающее меню появилось в крайнем правом положении горизонтального меню, для этого не обязательно программировать его динамически, достаточно лишь назвать его именем **MF_HELP**.

Давайте теперь посмотрим, как строится настоящая программа с меню **File** и **Edit**, которые нам уже знакомы.

Программа Stanmenu

На примере программы **Stanmenu** мы увидим, как необходимо компоновать отдельные ее части, чтобы наши меню заработали. Собственно меню определено в отдельном файле с именем **STANMENU.RC**. Но в нем определяются только типы данных. Для того чтобы подключить его к остальной программе и заставить работать, нужен некоторый код - имя меню, для которого в окне просмотра структуры данных отводится переменная **WNDCLASS**. Запишем одно из них в следующем виде:


```
wc.lpszMenuName="#1";
```

Строка "#1" означает, что для определения имени меню используют числовую константу. Таким образом проще и легче обратиться к нему. Можно использовать и стандартный код ASCII, но тогда мы займем больше памяти и затратим больше времени при работе.

Если Вы не хотите связывать меню с окном просмотра, то ссылку на него необходимо поместить в окне конструктора объектов в структуре Attr. Вы, вероятно, помните, что эта структура входит в состав данных, которые поступают в окно объектных классов из окна TWindow. Переменная Menu принимает значение LPSTR(char far*) имени меню. Например, возьмем меню и назовем его Short Menu. Чтобы задать его, необходимо в течение отведенного интервала времени записать в окно конструктора классов строку следующего вида:

```
Attr.Menu = "Short Menu"
```

А вот так будет выглядеть наша программа, создающая меню, Stanmenu:

MAKEFILE.MAK

AUTODEPEND

```
# Определения транслятора
INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;
C:\BORLANDC\INCLUDE
CC = bcc -c -D_CLASSDLL -H -ml -WS -w IS(INC)

# Неявные правила
.c.obj:
$(CC) {S< }
.cpp.obj:
$(CC) {S< }

# Явные правила
StanMenu.exe: StanMenu.res StanMenu.def StanMenu.obj
tlink /c/C/n/P-/Twe/x @StanMenu.LNK
RC StanMenu.res StanMenu.exe

# Зависимости отдельных файлов
StanMenu.obj: StanMenu.cpp
StanMenu.res: StanMenu.rc StanMenu.cur StanMenu.ico
RC -R -FO StanMenu.res StanMenu.RC
```

STANMENU.LNK

```
c:\borlandc\lib\cOwl.obj+
StanMenu.obj
StanMenu, StanMenu
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwl.lib+
import.lib+
mathl.lib+
cl.lib
StanMenu.def
```

STANMENU.H

```
// Определение кодов команд
#define IDM_FILE_NEW 22000
#define IDM_FILE_OPEN 22001
```



```

#define IDM_FILE_SAVE      22002
#define IDM_FILE_SAVEAS    22003
#define IDM_FILE_PRINT     22004
#define IDM_FILE_EXIT      22005
#define IDM_EDIT_UNDO      22006
#define IDM_EDIT_CUT       22007
#define IDM_EDIT_COPY      22008
#define IDM_EDIT_PASTE     22009
#define IDM_EDIT_CLEAR     22010
#define IDM_EDIT_DELETE    22011

// #define IDM_FILE_CLOSE  22012

```

STANMENU.CPP

```

/*
STANMENU.CPP Программа, демонстрирующая стандартные меню File и Edit
*/

```

```

#include <owl.h>
#include "stanmenu.h"

```

```

#define COMMANDMSG(arg) (arg.WParam)

```

```

/*
Описания класса
*/

```

```

class TStanMenuApplication : public TApplication

```

```

{
public:
    TStanMenuApplication(LPSTR lpszName, HANDLE hInstance,
                        HANDLE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow);
    virtual void InitMainWindow ();
};

```

```

class TStanMenuWindow : public TWindow

```

```

{
public:
    TStanMenuWindow(PWindowsObject pwParent, LPSTR lpszTitle,
                    PTModule pmModule);
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WNDCLASS&);
    virtual void WMCommand(TMessage& Msg) = [WM_COMMAND];
};

```

```

/*
Функция WinMain
*/

```

```

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
{
    TStanMenuApplication StanMenu ("StanMenu", hInstance, hPrevInstance,
                                    lpszCmdLine, nCmdShow);

    StanMenu.Run();
    return StanMenu.Status;
}

```



```

/*
Компонентная функция класса прикладной программы
*/
TStanMenuApplication::TStanMenuApplication
    (LPSTR lpszName, HANDLE hInstance,
     HANDLE hPrevInstance,
     LPSTR lpszCmdLine, int nCmdShow)
    :TApplication (lpszName, hInstance,
     hPrevInstance, lpszCmdLine, nCmdShow)
{
    /* Здесь инициализируется прикладная программа. */
}

/*
Компонентная функция класса прикладной программы
*/
void TStanMenuApplication::InitMainWindow ()
{
    MainWindow = new TStanMenuWindow(NULL, "A Standard Menu", NULL);
}

/*
Компонентная функция класса TStanMenuWindow
*/
TStanMenuWindow::TStanMenuWindow
    (PTWindowsObject pwParent, LPSTR lpszTitle, PTModule pmModule)
    :TWindow(pwParent, lpszTitle, pmModule)
{
    /* Здесь инициализируется окно */
}

/*
Компонентная функция класса TStanMenuWindow
*/
LPSTR TStanMenuWindow::GetClassName ()
{
    return "StanMenu:MAIN";
}

/*
Компонентная функция класса TStanMenuWindow
*/
void TStanMenuWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
    wc.lpszMenuName = "#1";
}

/*
Компонентная функция класса TStanMenuWindow
*/
void TStanMenuWindow::WMCommand(TMessage& Msg)
{
    char buffer[80];
    if (COMMANDMSG(Msg) == IDM_FILE_EXIT)

```



```

    SendMessage (HWindow, WM_SYSCOMMAND, SC_CLOSE, 0L);
else
{
    wsprintf(buffer, "Command = %d", COMMANDMSG(Msg));
    MessageBox (HWindow, buffer, "WM_COMMAND", MB_OK);
}
}

```

STANMENU.RC

```

#include "stanmenu.h"

snapshot icon StanMenu.ico

hand cursor StanMenu.cur

1 MENU
{
    POPUP "&File"
    {
        MENUITEM "&New",          IDM_FILE_NEW
        MENUITEM "&Open...",      IDM_FILE_OPEN
        MENUITEM "&Save",          IDM_FILE_SAVE
        MENUITEM "Save &As...",    IDM_FILE_SAVEAS
        MENUITEM SEPARATOR
        MENUITEM "&Print",         IDM_FILE_PRINT
        MENUITEM SEPARATOR
        MENUITEM "E&xit",          IDM_FILE_EXIT
    }
    POPUP "&Edit"
    {
        MENUITEM "&Undo\tAlt+Backspace", IDM_EDIT_UNDO
        MENUITEM SEPARATOR
        MENUITEM "Cu&t\tShift+Del",      IDM_EDIT_CUT
        MENUITEM "&Copy\tCtrl+Ins",       IDM_EDIT_COPY
        MENUITEM "&Paste\tShift+Ins",     IDM_EDIT_PASTE
        MENUITEM SEPARATOR
        MENUITEM "Cl&ear",                IDM_EDIT_CLEAR
        MENUITEM "&Delete",               IDM_EDIT_DELETE
    }
}

```

STANMENU.DEF

```

NAME STANMENU

EXETYPE WINDOWS

DESCRIPTION 'StanMenu - Sample Standard Menu'

CODE MOVEABLE DISCARDABLE

DATA MOVEABLE MULTIPLE

HEAPSIZE 512

STACKSIZE 5120

```

После выбора необходимых пунктов меню нашей программе направляется сообщение вида WM_COMMAND. Это сообщение изменяет переменную wParam, и теперь мы точно знаем, какой именно пункт меню выбран. Очевидно, для каждого пункта меню должно быть свое значение кода этой переменной, чтобы мы могли отличить их друг от друга. Делается это просто. В программе Stanmenu после прихода сообщения WM_COMMAND, например, все эти операции занимают не так уж много

места. Если полученный от меню итоговый код представляет сообщение вида `IDM_FILE_EXIT`, которое соответствует команде `Exit` в меню `File`, то мы останавливаем работу нашей программы, отсылая ей системное командное сообщение на закрытие главного окна прикладного уровня. В противном случае мы выводим на экран окно сообщений, в котором содержится меню, соответствующее этому коду.

В ответ на сообщение `WM_COMMAND` можно также использовать переключения типа `CASE` для каждого значения итогового кода. Помимо этого, Вы можете определить набор функций ответных сообщений, поставив в соответствие каждой команде свою функцию.

Если Вы помните, в главе 5 мы уже рассматривали функции - обработчики сообщений, которые входят в один из производных классов `TWindows`. Каждая из таких функций имеет свой индивидуальный номер, а для тех функций, которые имеют дело только с каким-либо одним сообщением, например, таким, как `WM_PAINT` или `WM_LBUTTONDOWN`, численное значение этого сообщения само является индивидуальным номером. Например, для `WM_PAINT` это значение равно 15, а для `WM_LBUTTONDOWN` - 513.

Для некоторых команд типа `WM_COMMAND` функции ответных сообщений определяют по верхней границе их значений, между величинами `0xA0000` и `0xFFFF`. Чтобы Вам было удобнее работать, в модуле `OWL` имеется файл `OWLDFSH`, в котором определена константа, указывающая, где именно находится эта граница. Приведем для примера описание класса `TStanMenu`, в котором для каждого пункта меню задана своя функция ответных сообщений:

```
class TStanMenuWindow : public TWindow
{
public:
    TStanMenuWindow (PTWindowsObject pwParent, LPSTR lpszTitle,
                    PTModule pmModule);
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WINDCLASS&);
    void CmdFileNew(TMessage& Msg) = (CM_FIRST+IDM_FILE_NEW);;
    void CmdFileOpen(TMessage& Msg) = (CM_FIRST+IDM_FILE_OPEN);;
    void CmdFileSave(TMessage& Msg) = (CM_FIRST+IDM_FILE_SAVE);;
    void CmdFileSaveAs(TMessage& Msg) = (CM_FIRST+IDM_FILE_SAVEAS);;
    void CmdFilePrint(TMessage& Msg) = (CM_FIRST+IDM_FILE_PRINT);;
    void CmdFileExit(TMessage& Msg) = (CM_FIRST+IDM_FILE_EXIT);;
    void CmdEditUndo(TMessage& Msg) = (CM_FIRST+IDM_EDIT_UNDO);;
    void CmdEditCut(TMessage& Msg) = (CM_FIRST+IDM_EDIT_CUT);;
    void CmdEditCopy(TMessage& Msg) = (CM_FIRST+IDM_EDIT_COPY);;
    void CmdEditPaste(TMessage& Msg) = (CM_FIRST+IDM_EDIT_PASTE);;
    void CmdEditClear(TMessage& Msg) = (CM_FIRST+IDM_EDIT_CLEAR);;
    void CmdEditDelete(TMessage& Msg) = (CM_FIRST+IDM_EDIT_DELETE);;
};
```

Каждая команда меню возвращает код типа `IDM_FILE_NEW`, соответствующий только ей одной, индекс таблицы соответствия для каждой функции ответного сообщения также будет соответствовать только ей одной, и Вы можете обрабатывать каждую команду, как если бы это было отдельное сообщение. На примере программы `Stanmenu` видно, что даже простое меню, создаваемое в среде `Windows`, требует определенных знаний и труда, создать его не так просто, как кажется. Однако если Вам вдруг захочется побольше узнать об их создании, например как в реальном времени изменять их, дополнять новыми записями, заменять в них надписи на полутонные и т. д., то к Вашим услугам в среде `Windows` предусмотрены еще более двух десятков подпрограмм поддержки меню. Всего их 26 в среде `Windows`, а чтобы Вам было легче разобраться с ними, разобьем их на шесть категорий и рассмотрим каждую из них в отдельности.

Подпрограммы поддержки меню

Как мы уже сказали, в среде Windows есть 26 подпрограмм поддержки меню, разделенных на шесть различных категорий, каждая из которых описывает тот или иной вид действий, который может выполняться объектами, входящими в меню. К ним относятся:

- создание надписи;
- установление связи с окном;
- уничтожение;
- модификация;
- запрос;
- захват.

Если Вам необходимо создать меню, в котором выполняются не только простые, но и сложные операции над объектами, то в любой группе из приведенных выше Вы можете найти то, что поможет Вам в решении задачи.

В табл.11.2 приведены краткие сведения о каждой из этих категорий.

Таблица 11.2. Краткие сведения о функциях поддержки меню.

Категория	Имя подпрограммы	Краткое описание
Создание	CreateMenu	Создает в памяти пустое меню
	CreatePOPUPMenu	Создает в памяти пустое всплывающее меню
	LoadMenu	Загружает меню из файла (EXE или DLL)
	LoadMenuDirect	Загружает меню из резервного меню, записанного в памяти
Установление связи с окном	SetMenu	Устанавливает, какому окну будет соответствовать данное меню верхнего уровня
Уничтожение	DeleteMenu	Удаляет пункт, а также все сопутствующие ему всплывающие меню из основного меню
	DestroyMenu	Уничтожает данное меню и все меню, находящиеся под ним
Модификация	AppendMenu	Дописывает дополнительный пункт в меню
	ChangeMenu	Аналогична функции модификации в версиях Windows 1.X и 2.X
	CheckMenuItem	Переключает состояние проверочного символа во всплывающем меню
	DrawMenuBar	Обновляет меню верхнего уровня после внесения в него изменений
	HiliteMenuItem	Переключает вид подсветки или выделения надписи пункта в меню верхнего уровня
	EnableMenuItem	Включает и отключает пункты в меню и изменяет их надписи на полутонные и обратно
	InsertMenu	Вводит новый пункт в меню
	ModifyMenu	Заменяет пункт в меню
	RemoveMenu	Удаляет всплывающее меню или пункты в меню. Удаление меню не означает, что его уничтожили. Им снова можно будет пользоваться
	SetMenuItemBitmaps	Устанавливает, что вместо принимаемого по умолчанию режима проверки дисплея или его отмены используются два битовых образа

Запрос	GetMenu	Передает меню в окно, с которым будет работать меню верхнего уровня
	GetMenuCheckMarkDimensions	Возвращает размер установленного по умолчанию значения проверочного символа меню управляющего программой дисплея
	GetMenuItemCount	Возвращает число занесенных в меню пунктов
	GetMenuItem	Определяет идентификационный номер данного пункта меню
	GetMenuState	Возвращает текущее состояние файлов в данном меню
	GetMenuString	Возвращает значение метки, соответствующей пункту меню
	GetSubMenu	Передает управление всплывающего меню данному меню
Захват	SetSystemMenu	Передает управление системному меню
	TrackPopupMenu	Создает всплывающее меню, которое может появиться в любой точке экрана

Для рядового пользователя меню представляет некий объект, расположенный в определенном окне. Вся работа, которую осуществляет среда Windows, остается для него как бы за кулисами. Но нас с Вами, как программистов, интересует именно эта работа, происходящая за кулисами. Мы должны ясно представлять все процессы, происходящие там, только тогда мы уверенно можем говорить, что все работает, как задумано.

С точки зрения программиста при определении внешнего вида меню и его функции прежде всего необходимо обеспечить его поддержку, для чего создаются определенные структуры данных. В большинстве случаев создатели подобного рода программ идут по пути наименьшего сопротивления и ресурсы меню передают окну класса. Тогда при создании такого окна, как мы уже упоминали, автоматически появляется соответствующее меню.

Однако Вашу программу можно связать более тесно с теми структурами данных, которые входят в меню. Мы уже видели, что в программе можно вызвать процедуру LoadMenu и загрузить с ее помощью ресурсы меню. А с помощью подпрограмм CreateMenu и CreatePopupMenu можно создавать пустые меню, которые впоследствии можно заполнить названиями команд или связать с другими всплывающими меню. Добавлять записи в меню можно с помощью любой из подпрограмм модификации. В примере, который мы сейчас приведем, будет использована подпрограмма AppendMenu. В этом фрагменте программы на основе знакомой уже нам структуры данных создается меню, подобное тому, которое было создано в его структуре в программе Stanmenu, однако здесь элементы более тесно связаны между собой.

```
{
HMENU hSub;
HMENU hTop;
hTop = CreateMenu();
hSub = CreatePopupMenu();
AppendMenu(hSub, MF_STRING, 1, "&NEW");
AppendMenu(hSub, MF_STRING, 2, "&OPEN");
AppendMenu(hSub, MF_STRING, 3, "&SAVE");
AppendMenu(hSub, MF_STRING, 4, "SAVE&AS...");
AppendMenu(hSub, SEPARATOR, 0, 0);
AppendMenu(hSub, MF_STRING, 5, "&PRINT");
AppendMenu(hSub, SEPARATOR, 0, 0);
}
```



```

AppendMenu(hSub,MF_STRING,6,"E&XIT");
hSub = CreatePopupMenu();
AppendMenu(hSub,MF_STRING,7,"UNDO\t Alt+Backspace");
AppendMenu(hSub,SEPARATOR,0,0);
AppendMenu(hSub,MF_STRING,8,"CU&T\t Shift+Del");
AppendMenu(hSub,MF_STRING,9,"&COPY\t Ctrl+Ins");
AppendMenu(hSub,MF_STRING,10,"&PASTE\t Shift+Ins");
AppendMenu(hSub,SEPARATOR,0,0);
AppendMenu(hSub,MF_STRING,11,"CL&EAR");
AppendMenu(hSub,MF_STRING,12,"&DELETE");
AppendMenu(hTop,MF_POPUP,hSub,'&EDIT');
SetMenu(hwnd,hTop);
}

```

Подпрограмма AppendMenu "склеивает" отдельные части меню в единое целое. С ее помощью в меню вводят название нового пункта и осуществляют связь между всплывающим меню и меню верхнего уровня (горизонтальным меню). Определяется эта подпрограмма следующим образом:

```

BOOL AppendMenu(hMenu, wFlags, wIDNewItem, lpNewItem)

```

здесь

hMenu задает всплывающее или горизонтальное меню;

wFlags - одна переменная MF_FLAGS или более (о них мы расскажем ниже);

wIDNewItem возвращает код поступившего сообщения WM_COMMAND или управляет всплывающим меню, если таковое дополняют в данное меню;

lpNewItem - переменная, в которую могут входить переменные трех типов.

При дополнении меню строка представляет переменную типа указатель на строку. Если меню дополняют битовым образом, то она представляет управляющий символ битового образа. И, наконец, если Вы хотите ввести в меню вместо надписи или битового образа специально разработанный Вами символ, то эта переменная идентифицирует Ваш символ и имеет удвоенную длину. Подпрограмму AppendMenu можно использовать по-разному в зависимости от того, с чем Вы имеете дело - с надписью в меню или со всплывающим меню, а также, будет ли включаемый в меню пункт представлять строку, битовый образ или символ, предложенный программистом. Для того чтобы дополнить Ваше меню строковой переменной, подпрограмму AppendMenu можно вызвать следующим образом:

```

AppendMenu(hMenu,MF_String,wID,"Open...");

```

Переменная wID принимает значение возвращаемого кода команды, передаваемого в сообщении WM_COMMAND при выборе пользователем пункта меню, который будет идентифицирован меткой "Open...".

Если же в Ваше меню верхнего уровня необходимо ввести выпадающее меню или в выпадающее меню нужно ввести еще одно, то эта команда будет вызываться в следующем виде:

```

AppendMenu(hMenuTop,MF_POPUP,hMenuPopup,"File");

```

И в этом случае третий параметр с именем hMenuPopup является уже не итоговым кодом команды, а управляющим символом всплывающего меню, который нужно дописать в конце меню, определенного этим параметром. Это меню носит название "File". Кроме строковой переменной в подпрограмме AppendMenu в качестве метки того, что отображено на экране для данного пункта меню, может использоваться битовый образ. В этом случае последний параметр в команде AppendMenu используется, чтобы вместо строкового указателя задать управляющий символ битового образа.

В следующем примере переменная `hbm`, входящая в состав макропараметра `MAKELONG`, стоящего последним в списке параметров, является управляющим символом битового образа:

```
AppendMenu(hMenu, MF_BITMAP, wID, MAKE LONG(hbm,0));
```

В флаговое поле `wFlags` подпрограммы `AppendMenu` входят 12 флагов, состояния которых могут принимать одно из двух возможных значений. Десять из них можно вызвать из ресурсного файла, а оставшихся два могут использоваться только в динамически создаваемых меню - это флаги `MF_BITMAP` и `MF_OWNERDRAW`. На примерах, которые будут приведены в главе 12, мы покажем, как их использовать. Краткое описание всех флагов приведено в табл.11.3. Их можно разбить на четыре категории, которые немного отличаются друг от друга. Список этих категорий и входящих в них флагов приведен в табл.11.4. Первым в каждом из этих списков приводится значение, принимаемое по умолчанию.

Таблица 11.3. Флаги создаваемых меню.

Флаг меню	Наличие флага	Краткое описание
<code>MF_BITMAP</code>	Нет	Выводит для пункта меню на экран битовый образ вместо строки. Это один из методов вывода изображений на экран. Второй - использование пункта меню под названием <code>MF_OWNERDRAW</code>
<code>MF_CHECKED</code>	Есть	Помещает проверочный символ вслед за пунктом меню
<code>MF_DISABLED</code>	Есть	Блокирует пункт меню. Вместо этого флага лучше использовать флаг <code>MF_GRAYED</code> , потому, что в этом случае пользователь видит, что данным пунктом воспользоваться нельзя
<code>MF_ENABLED</code>	Есть	Разблокирует пункт меню
<code>MF_GRAYED</code>	Есть	Блокирует пункт меню и делает его надпись полутонкой
<code>MF_MENUBARBREAK</code>	Есть	Осуществляет разделение меню, причем во всплывающих меню создает еще вертикальную разделительную черту
<code>MF_MENUBREAK</code>	Есть	Осуществляет разделение меню, но уже без вертикальной черты
<code>MF_OWNERDRAW</code>	Нет	Отсылает программе, создающей меню, сообщение типа <code>WM_DRAWITEM</code> , содержащее управляющий символ, который позволяет использовать при создании меток разрабатываемого меню контекст устройства для вызовов программы рисования элементов изображения графического дисплея
<code>MF_POPUP</code>	Есть	Подключает всплывающее меню к меню верхнего уровня или к всплывающему меню
<code>MF_SEPARATOR</code>	Есть	Создает горизонтальную разделительную черту в меню
<code>MF_STRING</code>	Есть	Задаёт в качестве метки пункта меню строковую переменную
<code>MF_UNCHECKED</code>	Есть	Изображение пункта меню создается без контрольной метки

Таблица 11.4. Категории флагов меню.

Флаги, определяющие тип объекта	Флаги, изменяющие режимы проверки	Флаги, вводящие, отменяющие блокировку	Флаги, осуществляющие разделение меню
MF_STRING	MF_UNCHECKED	MF_ENABLED	HET
MF_POPUP	MF_CHECKED	MF_GRAYED	MF_MENUBARBREAK
MF_BITMAP		MF_DISABLED	MF_MENUBREAK
MF_SEPARATOR			
MF_OWNERDRAW			

Наконец, осталась последняя подпрограмма для создания меню, которую мы еще не рассматривали, это процедура LoadMenuIndirect. Она создает меню в соответствии с шаблоном, который хранится в памяти машины. Действует она так же, как и процедура LoadMenu, за исключением того, что процедура LoadMenu считывает шаблон не из памяти машины, а с внешнего носителя - диска.

Итак, программа LoadMenuIndirect считывает шаблон из памяти, и, следовательно, Вы можете создавать шаблоны, так сказать, с налета, передавая их затем системному меню. При выполнении этой работы фактически требуется, чтобы Вы продублировали работу ресурсного компилятора по созданию объекта памяти, описывающего меню. Такие объекты или структуры данных в файлах WINDOWS.H получили название MENUITEMPLATEHEADER и MENUITEMPLATE.

Рассмотрим теперь подпрограмму SetMenu. Она единственная позволяет подключить меню к окну.

Подключение меню к окну

Размещением меню в окне управляет всего одна подпрограмма SetMenu. Этой программе соответствует целая категория, потому что именно она одна позволяет осуществлять замену меню верхнего уровня. Кроме того, как мы увидим, эта подпрограмма связана с очищением системной памяти. Синтаксис этой процедуры таков:

BOOL SetMenu(hWnd, hMenu);

где

* hWnd - управляющий символ окна WS_OVERLAPPED или окна WS_POPUP. Меню при этом нельзя подключать к окну WS_CHILD;

* hMenu - управляющий символ меню верхнего уровня, подключаемого к данному окну.

Ваша программа может создавать в ходе работы несколько различных меню и по желанию пользователя в различные моменты выводить их на экран. Это необходимо по ряду причин. В частности, уровень подготовки пользователей неодинаков, и для начинающих программистов необходимы лишь короткие меню с основными командами, а для тех, кто пользуется Вашей программой постоянно, следует создавать меню расширенными, позволяющими узнать написанную программу в деталях.

Это связано также с повышением уровня защиты Вашей программы. Так, например, можно использовать меню различных уровней, чтобы подключать различные уровни программы, для этого Ваша программа может запросить у пользователя пароль и создать нужное меню в соответствии с ним. Тот, кто пользуется Вашей машиной без привилегий, увидит не все меню, входящие в состав Вашей программы, а для пользующихся привилегиями будет предоставлено все.

И, наконец, создавать несколько различных меню необходимо для того, чтобы Ваша программа соответствовала стандарту MDI (Множественный документационный интерфейс) и поддерживала его. Этот стандарт, относящийся к стандартам

пользовательского интерфейса, открывает по запросу пользователя для каждого вновь разрабатываемого документа свое окно редактора, потому что для работы с различными видами документов могут понадобиться различные меню. Например, в программе разработки документов, носящей название Excel, фирмы Microsoft существует два типа документов:

- рабочие формы (Workshelts);
- карты (Charts).

Для каждого из этих типов создано свое меню. Пользователь всегда может выбрать то меню, которое ему необходимо. Но чтобы ни заставило Вас создать несколько различных меню, подпрограмма SetMenu всегда поможет быстро найти нужное.

Хотим, однако, предупредить Вас, что, как только меню покидает окно, среда Windows тут же о нем забывает, но память все равно расходует, и если Вы останавливаете работу программы, не уничтожая при этом окно, то до тех пор, пока Вы будете находиться в среде Windows, этот кусок памяти нельзя будет использовать. Таким образом, убирая меню из окна, следует сразу же убедиться, уничтожено оно или нет, в противном случае Ваша программа будет использовать системную память (в модуле сегмента данных USER) нерационально.

Следующий раздел посвящается тому, как надо уничтожать меню.

Примечание: если Вы хотите заменить меню в окне с помощью процедуры SetMenu, убедитесь, что управляющий символ прежнего меню остался без изменения, и только после того, как Ваша программа выйдет из среды, уничтожайте меню. Уничтожить его необходимо, потому что автоматическое уничтожение меню в Windows не предусмотрено.

Уничтожение меню

В среде Windows существуют две программы, предусмотренные для уничтожения меню и для освобождения соответствующих участков памяти, - DestroyMenu и DeleteMenu.

Подпрограмма DestroyMenu уничтожает те меню, которые не подключены к окнам. Если попытаться использовать ее для уничтожения меню, подключенного к окну, то Ваша программа будет разрушена. Определяется эта процедура так:

```
BOOL DestroyMenu(hMenu);
```

где hMenu - управляющий символ всплывающего меню или меню верхнего уровня, которое должно быть уничтожено. При этом будут уничтожены и все связанные с ним меню.

Приведем фрагмент программы, где удаляют меню из окна и уничтожают его.

```
HMENU hmenu;  
hmenu = GetMenu(hwnd); /* отыскивают управляющий символ меню */  
SetMenu (hwnd,NULL); /* удаляют меню из окна */  
DetstroyMenu(hmenu); /* уничтожают меню */
```

Вам, однако, не нужно проделывать эту процедуру с каждым из созданных меню. Вспомним, что меню уничтожается вместе с окном, к которому оно подключено.

Вторая подпрограмма для уничтожения меню носит название DeleteMenu. Она удаляет пункт из меню и вместе с этим пунктом уничтожает все подчиненные ему всплывающие меню. Память, занятая под меню, как и при использовании подпрограммы DestroyMenu, освобождается. Определяют DeleteMenu так:

```
BOOL DeleteMenu(hmenu, nPosition, wFlags);
```

где:

• * hMenu - управляющий символ меню верхнего уровня либо всплывающего меню;

* nPosition определяет пункт меню, который необходимо удалить. Значение этой переменной зависит от текущего значения последнего параметра wFlags;

* wFlags - выражение MF_BYPOSITION либо выражение MF_BYCOMMAND.

Если переменная wFlags приняла значение MF_BYPOSITION, это означает, что удаляемый пункт отыскивают по его относительному положению в меню. Для первого пункта оно равно нулю, второго - единице, третьего - двум и т.п. Значение MF_BYPOSITION удобно использовать при удалении всплывающих меню, у которых нет возвращаемого кода.

Приведем пример того, как удаляют всплывающее меню Edit из меню верхнего уровня в программе Stanmenu:

```
HMENU hmenu;  
...  
hmenu = GetMenu(hwnd);  
DeleteMenu(hmenu, 1, MF_BYPOSITION);  
DrawMenuBar(hwnd);
```

В следующем параграфе мы увидим, что для того, чтобы переписать меню, в которое вносились изменения, необходимо использовать процедуру DrawMenuBar.

Флаг MF_BYPOSITION используют и в том случае, когда необходимо удалить пункт из всплывающего меню, однако прежде необходимо передать управление в то меню, где записан этот пункт, используя процедуру GetSubMenu. Например, для того, чтобы удалить из меню Edit команду Copy, порядковый номер которой равен четырем, необходимо составить такую подпрограмму:

```
HMENU hmenu;  
HMENU hmenuEdit;  
hmenu = GetMenu(hwnd);  
hmenuEdit = GetSubMenu(hMenu, 1);  
DeleteMenu = (hMenuEdit, 3, MF_BYPOSITION);
```

Быстрее, для удаления пунктов из всплывающего меню использовать другое значение переменной wFlag - значение MF_BYCOMMAND. При этом нужный пункт удаляют, используя возвращаемый код команды, который дает Вам ссылку на любой пункт в иерархии меню по его положению относительно меню верхнего уровня.

Покажем, как можно удалить команду Copy в этом случае:

```
HMENU hmenu;  
...  
hmenu = GetMenu(hwnd);  
DeleteMenu(hmenu, 9, MF_BYCOMMAND);
```

О подпрограмме DeleteMenu мы узнали, что помимо 12 флагов, которые связаны с процедурой AppendMenu, существуют еще два - MF_BYCOMMAND и MF_BYPOSITION, которые связаны с процедурой DeleteMenu. Таким образом, общее их число равно 14. Эти флаги формируют основу, используя которую, Вы можете в дальнейшем построить меню, какое пожелаете.

Перейдем теперь к рассмотрению следующей совокупности подпрограмм, которые используют для изменения меню после того, как оно уже создано.

Внесение изменений в меню

Если Вы в своей программе создали меню и подключили его к какому-либо окну, то почему бы теперь не попробовать изменить его. Для этого в среде Windows предусмотрено десять подпрограмм, которые позволяют обращаться с Вашим меню так, как Вы этого хотите. Одна из них нам уже знакома - это подпрограмма AppendMenu, еще четыре изменяют структуру созданного меню - это подпрограммы ChangeMenu, InsertMenu, ModifyMenu и RemoveMenu. Процедуру ChangeMenu рассматривать не-

зачем, она была создана для первых версий среды Windows и поэтому работать с ней неудобно и сложно. К тому же ее вполне можно заменить остальными подпрограммами внесения изменений в меню, их еще называют подпрограммами модификации.

Подпрограмма InsertMenu задает параметры пункта меню или всплывающего меню, вводимого в создаваемое. Подпрограмма AppendMenu отличается от нее тем, что вводит новые пункты меню только после старых, т.е. дописывает их в конце.

Подпрограмма ModifyMenu изменяет уже созданный пункт в меню. Ее можно использовать, например, когда необходимо заменить строку, изменить итоговый код программы, вновь ввести в работу пункт меню, отмеченный серым полутонном, или, наоборот, написать полутонном и отменить этот пункт. Подпрограмма RemoveMenu отделяет всплывающее меню от меню верхнего уровня или от других таких же всплывающих меню. Эта процедура не затрагивает внутренней структуры меню, и его можно использовать далее. Конечно, если Вы удаляете пункт из меню, следует обязательно уничтожить его перед тем, как Ваша программа завершит работу, в противном случае этот участок системной памяти так и останется занятым.

Первую из этих подпрограмм InsertMenu определяют в следующем виде:

```
BOOL InsertMenu(hMenu, nPosition, wFlags, wID, lpNew);
```

где:

- * hMenu - управляющий символ для всплывающего меню или меню верхнего уровня;

- * nPosition - пункт в меню, перед которым нужно вставить новый пункт. Это может быть либо его относительным положением в меню, либо возвращаемым кодом в зависимости от того, какой флаг используется - MF_BYPOSITION или MF_BYCOMMAND;

- * WFlags - комбинация из 12 флагов для меню, с которыми мы уже познакомились, когда рассматривали подпрограмму AppendMenu. А два других флага - MF_BYPOSITION и MF_BYCOMMAND - указывают на способ определения его положения: wID - итоговый код, если Вы вводите новый пункт в меню, или управляющий символ, если вводите в него всплывающее меню. Предлагают этот код вместе с сообщением WM_COMMAND; lpNew - управляющая величина, которая принимает значения переменных трех типов. Если в меню дописывается строка, это указатель строки текста двойной разрядности. Если вместо строки вставляют растр, это управляющий символ раstra. И, наконец, если Вы создаете собственный пункт меню, эта величина определяет вводимый вами пункт.

Приведем пример того, как можно записать команду Close в меню File, помещая его под командой "Open...":

```
HMENU hmenu;  
HMENU hmenuFile;  
...  
hmenu = GetMenu (hwnd);  
hMenuFile = GetSubMenu (hmenu, 0);  
InsertMenu(hMenuFile, 2, MF_BYPOSITION, 13, "&CLOSE");
```

В этом фрагменте программы, чтобы определить место для нового символа, используют флаг MF_BYPOSITION, а так как мы вводим новый символ во всплывающее меню, то мы также должны передать в это меню и управляющий символ. Передают управляющий символ в меню верхнего уровня с помощью процедуры GetMenu, а в меню File - процедуры GetSubMenu, так как она является первым (нулевым) пунктом в этом меню верхнего уровня. После того как мы передали управляющий символ в скорректированное нами меню, процедура InsertMenu дописывает в него нашу команду и приписывает ей итоговый код, равный 13. Второй параметр определяет, что данный новый пункт меню необходимо поместить перед пунктом, итоговый код которого равен 2. Поскольку мы ведем отсчет от нуля, этот пункт будет третьим.

На рис.11.9 показано меню, в которое мы заново дописали пункт под названием Close.

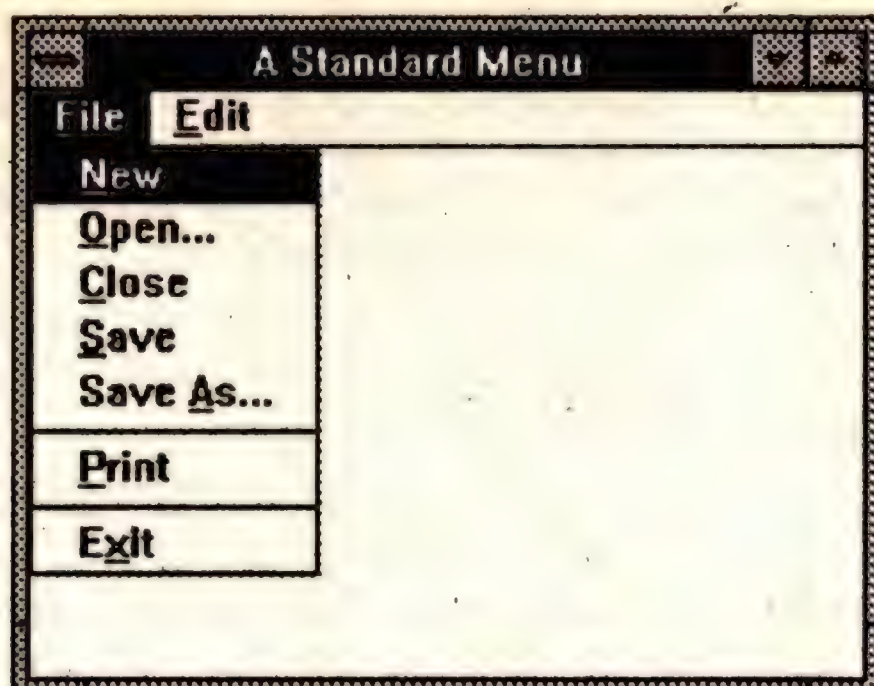


Рис.11.9. Меню с дописанным пунктом Close.

Попробуем теперь слегка изменить условия и дописать этот пункт на нужное место с помощью флага MF_BYCOMMAND. Этот способ проще, потому что вместо передачи управления на всплывающее меню Edit его передают только меню верхнего уровня, в которое предполагают записывать новый пункт:

```
HMENU hmenu;
...
hmenu = GetMenu(hwnd);
InsertMenu(hmenu, 3, MF_BYCOMMAND, 13, "&CLOSE");
```

В приведенном выше примере также дополняют меню с именем File пунктом Close. Но на этот раз его положение мы определяем с помощью флага MF_BYCOMMAND. В этом случае величина 3 равна итоговому коду пункта меню (его имя Save), перед которым мы собираемся поместить пункт Close.

Подпрограмма ModifyMenu, с помощью которой можно изменить все что угодно в пункте меню, имеет те же входные параметры и флаги, что и подпрограмма InsertMenu. В ней также определяют положение пункта меню с помощью флагов MF_BYPOSITION и MF_BYCOMMAND.

Понятно, что, поскольку эта подпрограмма вносит изменения в пункт меню, не изменяя самого меню, параметры этой подпрограммы имеют несколько иной смысл. В ней мы указываем пункт, в который будут вноситься изменения.

С помощью подпрограммы ModifyMenu можно устанавливать новую метку, соответствующую заданному пункту, включать или отменять его, изменяя тон его цвета. С помощью этой метки можно заменить обычный пункт меню на растр или на пункт, разработанный самим программистом, короче говоря, с помощью этой подпрограммы можно изменять в меню все, что введено в него с помощью подпрограммы InsertMenu и AppendMenu.

Приведем пример того, как с помощью подпрограммы ModifyMenu можно изменить названия в меню. Эти изменения проведем в меню Edit в пунктах Cut, Copy, Paste. Перепишем их названия на французском языке. Использовать при этом будем флаг MF_BYPOSITION:

```
HMENU hmenu;
HMENU hmenuEdit;
...
hMenu = GetMenu(hwnd);
hMenuEdit = GetSubMenu(hMenu, 1);
ModifyMenu(hMenuEdit, 2, MF_BYPOSITION, 8, "COUPER");
ModifyMenu(hMenuEdit, 3, MF_BYPOSITION, 9, "COPIER");
ModifyMenu(hMenuEdit, 4, MF_BYPOSITION, 10, "COLIER");
```


Итак, в приведенном примере мы с Вами смогли увидеть то, что когда используется флаг MF_BYPOSITION, то необходимо передавать управляющий символ в то меню, в котором мы хотим внести изменения. Когда используют флаг MF_BYCOMMAND, эта процедура упрощается.

Приведем еще один пример, как с ее помощью можно изменить названия пунктов меню, не прибегая к помощи процедуры SetSubMenu:

```
HMENU hmenu;  
...  
hMenu = GetMenu (hwnd);  
ModifyMenu (hmenu, 8, MF_BYCOMMAND, "COUPER");  
ModifyMenu (hmenu, 9, MF_BYCOMMAND, "COPIER");  
ModifyMenu (hmenu, 10, MF_BYCOMMAND, "COLIER");
```

Следующая функция, позволяющая вносить изменения в меню, - RemoveMenu. С ее помощью удаляют из меню ненужные пункты. Если это команды, то при их удалении освобождаются и соответствующие участки памяти. Если же это всплывающие меню, и они не уничтожены, можно использовать их вновь.

Определение функции RemoveMenu следующее:

```
BOOL RemoveMenu(hmenu,nPosition,wFlags);
```

где:

hMenu - управляющий символ всплывающего меню или меню верхнего уровня;
nPosition указывает, какой именно пункт меню надо удалить. Эта величина может также быть: либо относительным положением пункта в меню, либо его итоговым кодом в зависимости от того, какой выбран флаг - MF_BYPOSITION или MF_BYCOMMAND,

Используя флаг MF_BYCOMMAND, попробуем удалить из меню пункт Clear:

```
HMENU hmenu;  
...  
hmenu = GetMenu (hwnd);  
RemoveMenu (hmenu, 11, MF_BYCOMMAND);
```

Если бы мы захотели полностью удалить меню Edit, то нам следовало бы использовать флаг MF_BYPOSITION следующим образом:

```
HMENU hmenu;  
...  
hmenu = GetMenu (hwnd);  
hMenuEdit = GetSubMenu (hmenu, 1);  
RemoveMenu = (hmenu, 1, MF_BYPOSITION);  
DrawMenuBar = (hwnd);
```

Если мы не собираемся далее вновь подключать удаляемое меню к создаваемой нами программе, необходимо уничтожить его с помощью команды DestroyMenu(hMenuEdit) для того, чтобы освободить соответствующий участок памяти.

Вы, возможно, заметили, что после того, как всплывающее меню было удалено, мы вызвали процедуру DrawMenuBar. Ее нужно вызывать во всех случаях, когда мы вносим изменения в меню верхнего уровня, потому что подпрограммы, вносящие изменения в меню, изменяют только внутренние структуры данных, поддерживающие их. Чтобы пользователь мог увидеть внесенные изменения, меню необходимо нарисовать заново. Для этого и нужна функция DrawMenuBar.

Остальные подпрограммы, отнесенные к этой категории, используют для изменения состояния содержащихся в меню пунктов. Подпрограмма CheckMenuItem, например, предназначена для того, чтобы помещать или удалять галочку проверки вслед за пунктом меню. Кстати, если Вам не нравится, как она выглядит, то можно заме-

нить ее битовым образом, связать который с пунктом меню можно с помощью подпрограммы `SetMenuItemBitmap`. Последняя позволяет Вам получить битовый образ на экране даже тогда, когда пункт меню не проверялся.

Подробно о том, как это можно сделать, мы расскажем в главе 12, где коснемся вопроса о том, как улучшить внешний вид надписей в меню, используя графические процедуры.

Процедура `EnableMenuItem` позволяет выбрать одно из трех возможных состояний каждого пункта меню: "Включен", "Выключен" и "Изображен полутон". Как мы уже говорили, состояние "Выключен" из-за отсутствия обратной связи с пользователем лучше не применять. Его можно заменить состоянием "Изображено полутон". В этом случае пользователю сразу видно, что данный пункт меню выключен, и он его уже не выберет. Состояние "Включен" обычно принимают по умолчанию и изменяют по необходимости. И, наконец, последняя подпрограмма - `HiliteMenuItem`. Она позволяет делать ярче пункты, входящие в меню верхнего уровня. Эту процедуру вызывают с помощью кодовой комбинации клавиш, при нажатии которых надпись в меню делается ярче. Если Вы не собираетесь вызывать эту процедуру вручную, то она Вам не понадобится.

Следующая категория подпрограмм обрабатывает запросы системного меню для получения той или иной информации.

Запросы

Подпрограммы запроса позволяют получить необходимую информацию о меню. Существует два типа таких подпрограмм: подпрограммы первого типа предоставляют доступ к меню, а второго запрашивают информацию об атрибутах. И в том, и в другом случае, если Вы хотите установить значение величины или состояние флага, зависящие от состояния пункта меню, подпрограмма запроса помогает узнать, каково текущее значение этой величины или состояние этого флага.

Ключ меню, число, с помощью которого Вы имеете доступ к меню, можно узнать с помощью трех следующих подпрограмм: `GetMenu`, `GetSubMenu` и `GetSystemMenu`.

На рис.11.10 наглядно показано, о каком меню возвращает информацию каждая из этих процедур.

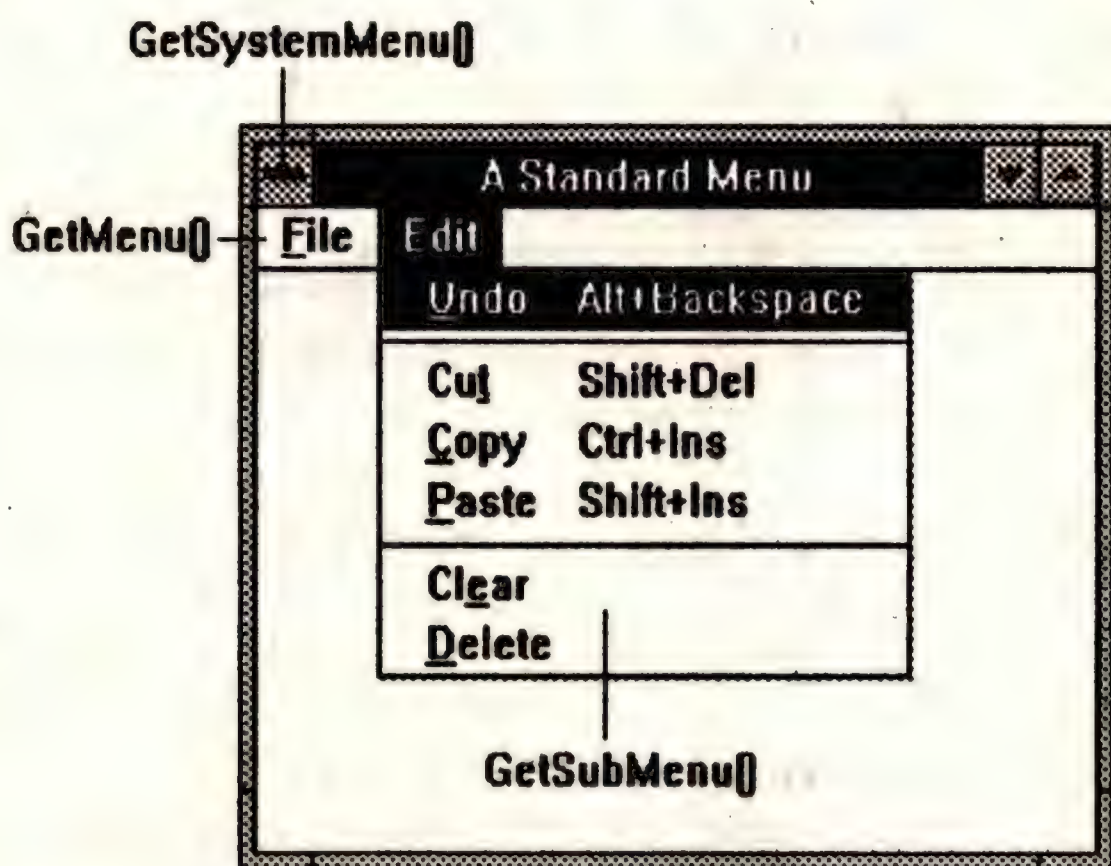


Рис 11.10. Процедуры, возвращающие информацию об управлении в меню.

Ключ меню верхнего уровня можно узнать с помощью процедуры `GetMenu`. Ее вызывают следующим образом:

```
hmenuTop = GetMenu(hwnd);
```


Если Вы знаете дескриптор меню верхнего уровня, то дескриптор всплывающего меню можно узнать, вызвав процедуру

```
hmenu = GetMenu (hmenuTop,nPosition);
```

где nPosition - порядковый номер всплывающего меню в меню верхнего уровня или в другом всплывающем меню, считая с нуля. Если Вы определили его заголовок, то можно вносить любые изменения: дополнять, удалять или изменять его пункты.

Подпрограмма GetSystemMenu позволяет по-разному использовать особенности системного меню, вызывая его двумя разными способами. Как и в большинстве программ в среде Windows, предусмотрена только одна копия системного меню. Для того чтобы создать ее в данном окне, необходимо воспользоваться командой GetSystemMenu следующего вида:

```
hmenuSys = GetSystemMenu(hwnd,0);
```

Нулевое значение второго параметра сообщает системному меню о том, что заголовок создаваемого системного меню попал к Вам, и Вы будете вносить в это меню свои изменения. Но, если после того, как изменили его, Вы решаете вернуться вновь к исходному меню, то следует вызывать его следующим образом:

```
hmenuSys = GetSysMenu(hwnd,1);
```

Отличное от нуля значение второго параметра подключит к Вашему окну системное меню с теми параметрами, которые были приняты по умолчанию, в это меню Вы уже не сможете внести изменения. Остальные пять подпрограмм запроса возвращают значения атрибутов меню. На рис.11.11 приведены те атрибуты, значения которых возвращают четыре подпрограммы.

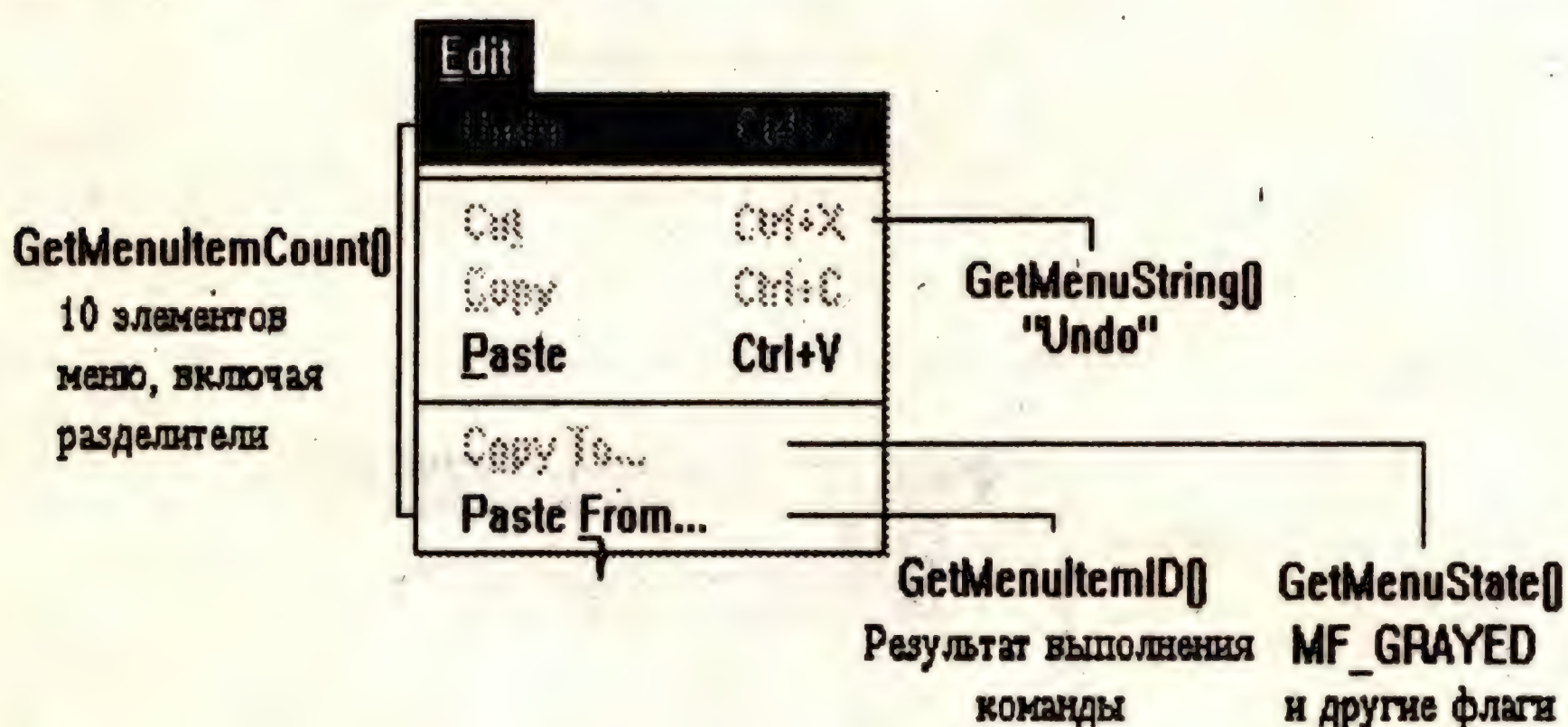


Рис.11.11. Атрибуты меню, узнать о состоянии которых можно с помощью программ запроса.

Определить, сколько пунктов входят в меню, можно вызовом процедуры

```
W = GetMenuItemCount(hMenu);
```

где hMenu - дескриптор либо всплывающего меню, либо меню верхнего уровня.

В число пунктов меню включается и вертикальная разделительная черта. Если Вам необходимо определить итоговый код команды пункта меню, воспользуйтесь подпрограммой:

```
W = GetMenuItemID(hMenu,nPosition);
```


Узнать о состоянии всплывающего меню с помощью этой процедуры не удастся, Вы получите в любом случае возвращаемую величину, равную 1. Чтобы определить дескриптор всплывающего меню, Вам нужна процедура GetSubMenu.

С помощью следующей процедуры GetMenuState Вы можете определить состояние флагов меню. Если Вам необходимо узнать состояние какого-либо одного флага, попробуйте воспользоваться функцией AND. Например, в следующем фрагменте программы проверяют состояние флагов для пункта меню, итоговый код которого равен 38, а затем определяют, изображен он полутонем или нет:

```
WORD wFlags;  
...  
wFlags = GetMenuState(hmennu,38,MF_BYCOMMAND);  
if (wFlags & MF_GRAYED)  
{  
.  
.  
.  
}
```

Однако, при использовании этой процедуры, будьте внимательны. Состояния всех флагов, принятые по умолчанию MF_STRING, MF_VNHLITE, MF_ENABLED, MF_UNCHECKED, равны нулю и проверить их с помощью функции AND невозможно. Поэтому проверяют не состояния данных флагов, а флагов, которые контролируют противоположные состояния, например, флаг MF_ENABLED контролирует включение пункта меню, а флаги MF_DISABLED и MF_GRAUED - его отмену, т.е. противоположное состояние, тогда программа примет вид:

```
WORD wFlags;  
...  
wFlags = GetMenuState(hmenu,38,MF_BYCOMMAND);  
if (!(wFags & (MF_DISABLED | MF_GRAYED))  
{  
.  
.  
}
```

Если ни флаг MF_DISABLED, ни флаг MF_GRAYED не подняты, то условное выражение возвращает значение TRUE. Такая проверка состояния флага выглядит несколько неуклюже, но, к сожалению, по-другому ее провести нельзя, поскольку состояние флага, принимаемое по умолчанию, записано в памяти.

Заметим еще, что в процедуре GetMenuState, чтобы найти нужный пункт меню, можно использовать как флаг MF_BYCOMMAND, так и флаг MF_BYPOSITION.

Если пункт представлен в меню строковой переменной (а не разделительной чертой, растром или символом пользователя), то скопировать его можно с помощью процедуры GetMenuString:

```
cbSize = GetMenuString(hmenu,wID,lpBuff,bufsize,wFag);
```

где

hMenu - дескриптор меню;
wID - либо абсолютное, либо относительное положение пункта в меню;
lpBuff - указатель буфера строковой переменной увеличенной разрядности;
bufsize - размер этого буфера;
wFlag - либо флаг MF_BYCOMMAND, либо флаг MF_BYPOSITION.

Значение величины cbSize равно числу скопированных битов. И, наконец, последняя процедура запроса, которая в среде Windows носит самое длинное название, - GetMenuCheckMarkDimensions. Она возвращает значение ширины и высоты галочки в меню (рис.11.12).

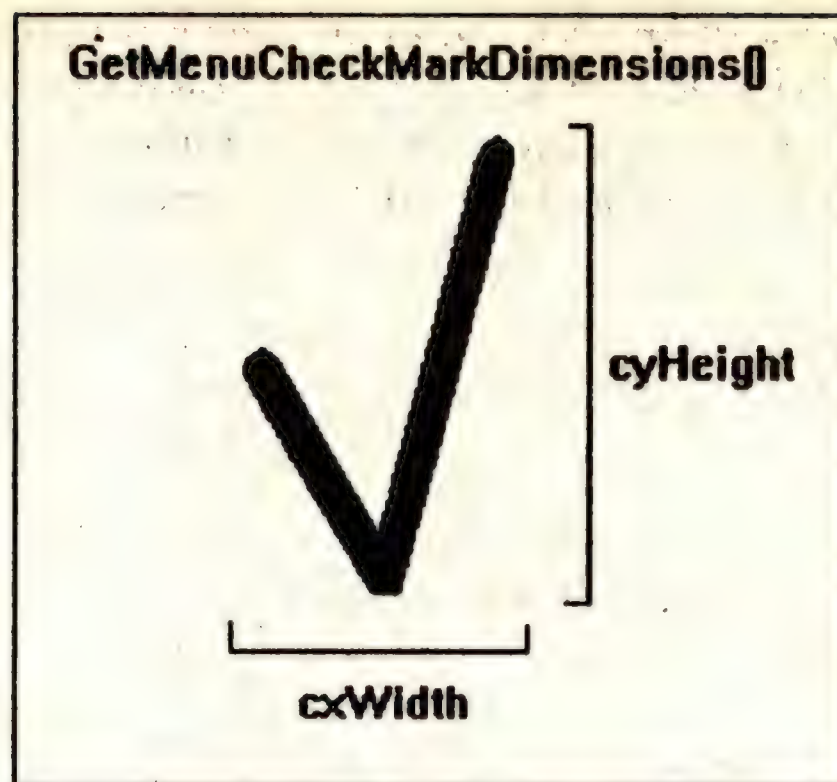


Рис.11.12. Размеры галочки.

Используют эту процедуру в тех программах, где принимаемый по умолчанию вид галочки заменяют иным, нужным автору. Установить ее вид можно и в случае, если пункт меню не проверяется. Процедура возвращает значение четырех битов, которые можно разбить на две части, одна из которых возвращает значение ширины, а другая — высоты галочки.

```
WORD dwCeck;
int cxWidth;
int cyHeight;

dwCheck = GetMenuCheckMarkDimensions();
cxWidth = LOWORD(dwCheck);
cyHeight = HIWORD(dwCheck);
```

Перейдем теперь к рассмотрению последней категории подпрограмм, поддерживающих меню вне окна, к которому оно присоединено, так называемых подпрограмм захвата.

Подпрограммы захвата

Эта категория представлена единственной подпрограммой `TrackPopupMenu`, позволяющей создать всплывающее меню в любом месте экрана. Иногда такие меню, не подсоединенные к меню верхнего уровня, называют изолированными. Выглядят они как обычные выпадающие меню, за исключением того, что могут находиться в любом месте экрана. Определяется процедура `TrackPopupMenu` в следующем виде:

```
BOOL TrackPopupMenu(hMenu,O,X,Y,O,hWnd,OL);
```

где

`hMenu` - дескриптор всплывающего меню;

`X` - координата `X` левого верхнего угла меню на экране дисплея;

`Y` - координата `Y` этого угла;

`hWnd` - дескриптор окна, в которое передаются сообщения `WM_COMMAND` и другие;

второй, пятый и седьмой параметры - резервные и обязательно равны нулю.

Для этой процедуры важно пояснить, что меню будет располагаться на экране в соответствии не с координатами рабочей области Вашей программы, а - с координатами экрана дисплея. Масштаб этих координат определяется размером его элементарного изображения - точкой. Координаты объекта (0,0) соответствуют его положению в левом верхнем углу экрана. Для того чтобы поместить Ваше меню в нужное место

рабочей области, необходимо привести его координаты, заданные для рабочей области, к координатам экрана. Процедура ClientToScreen значительно поможет в этом.

Приведем пример того, как сообщение от левой клавиши мыши можно использовать для переключения пункта в изолированном меню, на котором пользователь установил указатель:

```
WM_LBUTTONDOWN;  
{  
    HMENU hmenu;  
    HMENU hmenuEdit;  
    POINT pt;  
  
    hmenu = GetMenu(hwnd);  
    hmenuEdit = GetSubMenu(hmenu,1);  
    pt = MAKEPOINT(lpParam);  
    ClientToScreen(hwnd,&pt);  
    TrackPopupMenu(hmenuEdit, /* Дескриптор всплывающего меню */  
        0, /* Резервный параметр */  
        pt.x, /* Координата x */  
        pt.y, /* Координата y */  
        0, /* Резервный параметр */  
        hwnd, /* Дескриптор окна для пересылки сообщения */  
        0L); /* Резервный параметр */  
} break;
```

В этом фрагменте программы мы помещаем созданное в программе Stanmenu меню в любом месте экрана.

Всплывающее меню, созданное с помощью процедуры TrackPopupMenu, имеет такой же вид, как и обычное меню. Однако у него есть одна важная особенность, которая заключается в том, что сообщение WM_INITMENUPOPUP до появления меню на экране не появляется. Единственным сообщением об инициализации меню является сообщение типа WM_INITMENU. Это связано с тем, что изолированное меню в иерархии занимает место меню верхнего уровня, т.е. с точки зрения системного меню у него тот же статус, что и у заголовка окна, который помещается непосредственно под линией, очерчивающей это окно. Для инициализации меню верхнего уровня, в том числе и тех меню, которые созданы с помощью процедуры TrackPopupMenu, зарезервировано специальное сообщение типа WM_INITMENU. Но если изолированное меню, созданное с помощью процедуры TrackPopupMenu, должно иметь дополнительно всплывающие окна, и необходимо инициализировать эти окна, Вам направляют сообщение типа WM_INITMENUPOPUP.

Меню позволяют показать пользователю те команды, которые применяются в Вашей программе, однако некоторые пользователи предпочитают вместо меню работать с кодовыми комбинациями клавиш. Этим пользователям можно помочь - среда Windows позволяет создавать такие кодовые комбинации. Они позволяют осуществить непрерывную связь пользователя с машиной, не отрывая его на вызов меню. Зачастую при этом на короткое время высвечивают название вызываемой команды, чтобы пользователь знал, какую команду он вызывает.

Давайте посмотрим, какие возможности представляют нам кодовые комбинации клавиш и как их можно задавать в программе, создаваемой в среде Windows.

Кодовые комбинации клавиш

Первая версия Windows появилась в 1985 году, и разработка кодовых комбинаций клавиш, с помощью которых значительно ускоряется общение пользователя с программной средой, завершила для фирмы Microsoft всю работу по ее созданию.

Сначала ввод данных осуществлялся в основном с помощью меню, затем разработчики решили доработать программу в соответствии с возможностями существовавшей компьютерной техники. На их решение в большей степени повлияло мнение тех

программистов, которые разрабатывали программы в DOS, большей частью основанные на работе с клавиатурой. Эти программисты составляли значительную часть потенциальных потребителей программы Windows, и они в большей степени были заинтересованы в том, чтобы использовать ее в своей работе.

С помощью кодовых клавиш можно осуществлять ввод как с клавиатуры, так и через меню. Они значительно сокращают усилия пользователей, так как с помощью кодовых клавиш формируются точно такие же команды, которые используются в работе меню и которые мы в этой главе уже рассматривали.

Можно сказать, что фирме Microsoft удалось создать очень удачную вещь, потому что зачастую программисты обращают на использование кодовых клавиш не больше внимания, чем, скажем, на дополнения к основным меню. Но на самом-то деле кодовые клавиши представляют собой самостоятельную часть среды Windows и позволяют вызывать те команды, которые в меню отсутствуют.

В принципе можно даже обойтись и без меню, используя только кодовые клавиши, но нам кажется, что делать этого не следует, потому что использование меню и клавиш значительно упрощает работу пользователя.

В главе 15 мы узнаем, что трансляция команд, сформированных с помощью кодовых клавиш, происходит в два этапа. Аппаратная часть компьютера формирует вначале так называемые коды опроса для того, чтобы получить информацию о работе клавиатуры. Каждая клавиша формирует два кода: один говорит о том, что она нажата, другой, - что ее отпустили. С помощью кода опроса определяют, какие клавиши нажаты. Этот код малоинтересен большинству программ, потому что, во-первых, относится к сигналам, а во-вторых, в среде Windows его преобразуют в другой, более удобный вид информации.

На первом этапе трансляции коды опроса клавиши преобразуют в коды виртуальных клавиш. Эти коды ближе к используемым при формировании символов в программе, но с их помощью все еще нельзя, например, отличить заглавную букву от строчной. Это связано с тем, что коды виртуальных клавиш соответствуют не символам, а клавишам. С этой точки зрения они позволяют облегчить понимание процесса ввода информации с клавиатуры. Например, для них нет разницы, была ли нажата клавиша CapsLock в тот момент, когда пользователь набрал комбинацию Ctrl+A. Другими словами, они не позволяют различить, какую букву А мы набрали в этой комбинации - заглавную или строчную.

На втором этапе трансляции коды виртуальных клавиш преобразуют в расширенные коды ASCII - те самые, которые используются в среде Windows. Программа, например текстовый редактор, распознавая символ, понимает его только когда он записан в коде ASCII, а коды опроса и коды виртуальных клавиш ей непонятны. Это связывает нас: при работе с текстовым редактором мы не можем использовать часть клавиш, поскольку они задействованы под команды. Правда, если Вы пишете программу, которая будет использовать клавиатуру исключительно для ввода команд, то коды ASCII Вам не нужны.

Среда Windows позволяет назначать как клавиши, использующие коды виртуальных клавиш (такие клавиши еще называют функциональными), так и комбинации типа Alt+B или Ctrl+C, которые используют символы кода ASCII. Чаще всего в программах назначают функциональные клавиши, потому что они не зависят от состояния клавиш Shift. Но если Вы хотите создать программу, которая учитывала бы состояние регистров клавиатуры, то придется пользоваться кодами ASCII.

В программе, где используют кодовые клавиши, необходимо составить для них таблицу быстрого вызова команды. В программе можно использовать несколько таких таблиц, но работать они должны в разное время. Для того чтобы задать таблицу быстрого вызова команды, надо войти в ресурсный файл по ключевому слову ACCELERATORS. На рис.11.13 показано, что программа Resource Workshop, входящая в Windows, имеет встроенный редактор кодовых клавиш. Он позволяет обеспечить процесс задания таблицы быстрого вызова, вместо нее можно использовать любой текстовый редактор.

Приведем два возможных варианта задания таблиц быстрого вызова. Каждая строка в таблице соответствует своей кодовой комбинации.

```
table name ACCELERATORS;
{
<key>,<cmd>,VIRTKEY,[,NOINVERT][,ALT][,SHIFT][,CONTROL]
<key>,<cmd>,ASCII,[,NOINVERT][,ALT][,SHIFT][,CONTROL]
}
```

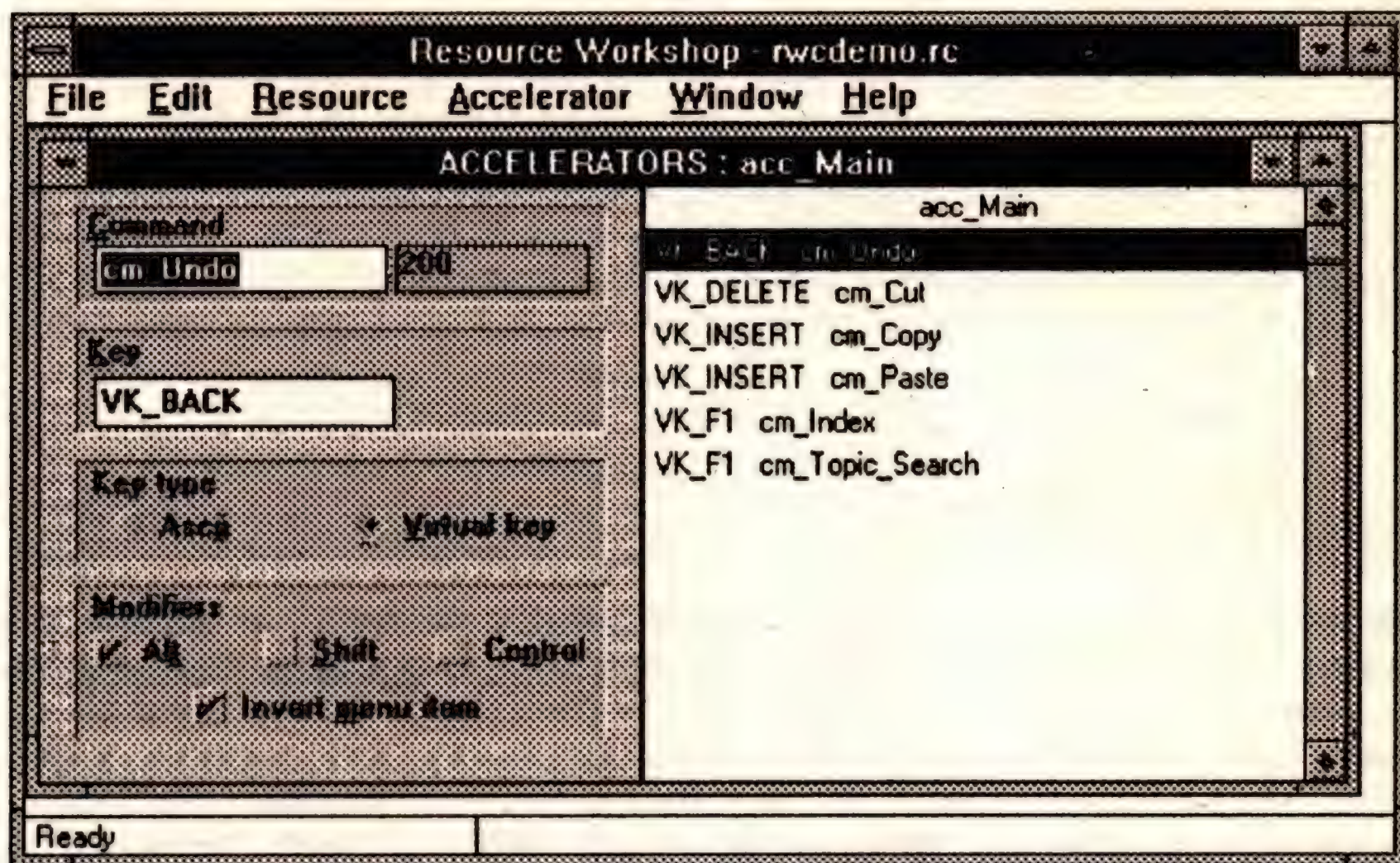


Рис.11.13. Программа Resource Workshop, позволяющая обеспечить задание кодовых клавиш.

<key> - либо клавиша текста в кавычках "A", либо константа виртуальной клавиши WINDOWS.H, например VK_F1, либо числовая переменная, например 58 или 0x6C;

<cmd> - итоговый код команды, который включается в сообщение типа WM_MESSAGE для того, чтобы сообщить программе, что нажата данная комбинация клавиш;

VIRTKEY - код нажатой клавиши <key> представляет код виртуальной клавиши;

ASCII - код нажатой клавиши окажется кодом ASCII;

NOINVERT препятствует автоматическому выделению соответствующей надписи команды в меню;

ALT - в данной кодовой комбинации используется клавиша Alt;

SHIFT - в данной кодовой комбинации используется клавиша Shift;

CONTROL - в данной кодовой комбинации используется клавиша Control.

Приведем таблицу, в которой используются только коды ASCII:

```
AscKeys ACCELERATORS;
```

```
{
"A",25,ASCII
"a",26,ASCII
"1",27,ASCII
}
```

Итоговый код команды - это величина, значение которой передается в составе сообщения WM_COMMAND параметру wParam оконной процедуры. Приведем пример; в котором эта ответная функция сообщения обеспечивает внутреннее прерывание

и обработку кодов, формируемых при нажатии клавиш в соответствии с данной таблицей быстрого вызова команды:

```
void TSampleWindow::WM_COMMAND(TMessage & Msg);
{
    switch(COMMANDMSG(msg))
    {
        case 25:
            .
            .
            .
        case 26:
            .
            .
            .
        case 27:
            .
            .
            .
    }
}
```

Таким образом Вы можете создавать программы, в которых для вызова команд используются обычные клавиши. Заметим заодно, что в этом случае Ваша программа начинает реагировать на состояние регистров клавиатуры. Вызов команды можно осуществить при нажатии клавиши Shift либо при включенном регистре CapsLock. Если же их нажать одновременно, то, как Вы знаете, они взаимно уничтожат друг друга, и вместо определенного действия на экране появится буква, которую Вы используете в этой комбинации с Shift.

В большинстве программ, однако, применяют функциональные клавиши и, соответственно, используют коды виртуальных клавиш, которые не зависят от состояний регистров. Эти клавиши также можно использовать в комбинации с клавишами Alt, Shift, Ctrl. Приведем в качестве примера еще одну таблицу быстрого вызова команды, в которой использованы функциональные клавиши:

```
VirtKeys ACCELERATORS;
{
    "A",35,VIRTKEY,CONTROL
    VK_F1,36, VIRTKEY
    VK_F8,37, ALT,CONTROL,SHIFT
}
```

Эта таблица задает три кодовых комбинации: Ctrl+A, F1 и Alt+Ctrl+Shift+F8. Так как функциональные клавиши формируют коды виртуальных клавиш, а не коды ASCII, то их работа не зависит от того, в каком положении находятся регистры. Например, при нажатии клавиш Ctrl+A безразлично, включен или выключен регистр CapsLock. Благодаря этому функциональные клавиши доставляют меньше хлопот пользователям. Вторая кодовая комбинация - функциональная клавиша F1 - используется, чтобы передать сообщение WM_COMMAND со значением переменной wParam, равным 36. И, наконец, последняя кодовая комбинация состоит из четырех клавиш: Alt, Ctrl, Shift и F8. Конечно, такую комбинацию задавать не стоит, потому что пользователю будет неудобно работать с ней, хотя, возможно, зная, как ее задавать, Вы сможете ночью спать спокойнее.

Некоторые клавиши в кодовых комбинациях задавать нельзя. Они приведены ниже:

```
AvoidThese ACCELERATORS;
{
```


VK_TAB, 99, VIRTKEY, ALT;	/*Переход с программы на программу */
VK_SPACE, 100, VIRTKEY, ALT;	/* Системное меню */
VK_F10, 101, VIRTKEY ;	/* Клавиша вызова меню */
VK_F4, 102, VIRTKEY, ALT ;	/* Заккрыть окно */
VK_ESCAPE, 103, VIRTKEY, CONTROL;	/* Вызов списка заданий */
VK_ESCAPE, 104, VIRTKEY, ALT;	/* Переход с программы на программу */
VK_ALT, 105, VIRTKEY;	/* Клавиша Alt */
VK_SHIFT, 106, VIRTKEY;	/* Клавиша Shift */
VK_CONTROL, 107, VIRTKEY,	/* Клавиша Ctrl */
VK_DELETE, 108, VIRTEY, CONTROL, ALT;	/* Перегрузка системы */

Первая категория (с кодами 99-104) включает те комбинации клавиш, которые в среде Windows зарезервированы и задействование которых может разрушить весь пользовательский интерфейс. Вторая категория включает клавиши Alt, Ctrl и Shift, которые по одиночке, вне комбинаций с буквами или функциональными клавишами, не используются, и, наконец, последняя комбинация не требует иного комментария, - ее ни в коем случае нельзя использовать.

Использовать некоторые клавиши лучше всего только для тех действий, для которых они зарезервированы, например комбинация Shift+Del в программах, использующих буфер вырезанного изображения, осуществляет вызов пункта в меню Edit.

В табл.11.5 приводятся краткие сведения о таких клавишах.

Таблица 11.5. Резервные кодовые комбинации клавиш.

Комбинация клавиш	Описание
Alt+Backspace	Пункт Undo в меню Edit
Shift+Del	Пункт Cut в меню Edit, служит для перемещения пункта в Clipboard
Ctrl+Ins	Пункт Copy в меню Edit копирует и передает пункт в Clipboard
Shift+Ins	Пункт Paste в меню Edit копирует содержимое Clipboard в документ
Del	Соответствует пунктам Clear или Delete в меню Edit, которые удаляют данные без изменения содержимого Clipboard
F1	Помощь
F6	Переход из одного текущего окна в другое
Ctrl+F6	Переход в другое окно в соответствии со стандартом MDI

Если Вы создаете программу в среде Windows и намереваетесь в будущем использовать ее в другой среде, то, очевидно, следует избегать комбинаций клавиши Alt с буквенными, например Alt+A. Они могут совпасть с мнемониками этих сред и вызвать конфликт в программе.

Как Вы помните, мнемониками мы ранее называли комбинации Alt+буква, которые вызывают то или иное меню. Если задать идентичную ей кодовую комбинацию в программе, это вызовет ее отмену, потому что кодовые комбинации имеют более высокий приоритет.

На первый взгляд обойти эту трудность не так уж сложно, рассмотрим для примера программу, в которой создаются два меню - File и Edit. Эти меню можно вызвать с помощью мнемоник Alt+F и Alt+E соответственно. Можно подумать, что использование комбинации Alt+T не вызовет нежелательных последствий. Однако, если вдруг Вашу программу необходимо будет использовать, например, в Финляндии, где слово File переводится как Tiedosto, а Edit как Muokkaus, то окажется, что его мнемоника войдет в конфликт с Вашей Alt+T. На рис.11.14 приведен пример того, как будет выглядеть программа в финском варианте.

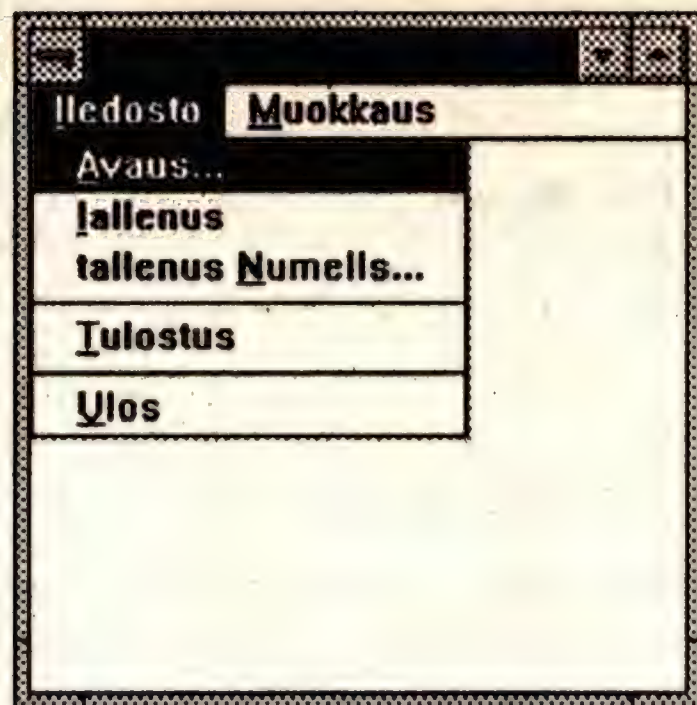


Рис.11.14. Пример программы на финском языке.

Из рис.11.14 видно, что в финском варианте кодовые комбинации Alt+T и Alt+M задавать нельзя. Проще всего решить эту проблему, исключая из использования комбинации типа Alt и одна буква.

После того как таблица кодовых комбинаций задана в ресурсном файле, ее передают в действующую программу с помощью процедуры LoadAccelerators. Она определяется следующим образом:

```
HANDLE LoadAccelerators(hInstance,lpTableName);
```

где

hInstance - дескриптор, передаваемый в программу как и параметр программы WinMain;

lpTableName - длинный указатель строковой переменной, содержит имя таблицы ускоренного вызова в ресурсном файле программы.

Возвращаемое процедурой LoadAccelerators значение - это дескриптор той таблицы ускоренного вызова, в которой определены заданные кодовые комбинации клавиш.

Давайте попробуем загрузить таблицу ускоренного вызова с именем VIRTKEYS:

```
HANDLE hAccel
hAccel = LoadAccelerators(hInstance,"VIRTKEYS");
```

Для того чтобы задать таблицу быстрого вызова, можно использовать что угодно, в том числе меню, о которых мы только что говорили, или числовые величины, например:

```
23 ACCELERATORS
{
    "A",35,VIRTKEY,CONTROL
}
.
```

Загружая данную таблицу в память, необходимо вызвать процедуру LoadAccelerators:

```
hAccel = LoadAccelerators(hInstance, "#23");
```

либо использовать макропроцедуру MAKEINTRESOURCE, с ее помощью можно упаковать эту табличку таким образом, чтобы она выглядела как числовая величина:

```
hAccel = LoadAccelerators(hInstance, MAKEINTRESOURCE(23));
```


После того как таблица загружена, для ее использования надо в цикле передачи сообщений вызвать процедуру TranslateAccelerator. Поскольку она уже вызвана с помощью объекта OWL уровня применений, не будем углубляться в ее изучение.

Как мы уже говорили, кодовые комбинации клавиш и меню формируют совершенно одинаковые сообщения. Одно из них - WM_COMMAND - переносит команды к объекту окна. Как и в случае с командами меню, Вы можете задать для каждой команды единственную функцию ответа на сообщение. В этом случае вначале убедитесь в том, что величина CM_FIRST соответствует индексу первой командной функции.

Трансляция кодовых комбинаций клавиш

С помощью процедуры TranslateAccelerators коды, возникающие при наборе кодовых комбинаций, можно транслировать в соответствующие коды меню автоматически. Необходимо лишь включить эту процедуру в цикл передачи сообщений. Ее обычно вызывает для нас объект прикладного уровня в модуле OWL.

Процедуры TranslateAccelerators определяют следующим образом:

```
int TranslateAccelerators(hwnd,haccTable,lpMsg);
```

где

hwnd - дескриптор окна, куда отсылается данная команда и управляющие сообщения для меню, создаваемые с помощью данной комбинации;

haccTable - дескриптор таблицы быстрого вызова, загруженной с помощью процедуры LoadAccelerators;

lpMsg - указатель длины для структуры данных типа MSG.

Чаще всего кодовые комбинации клавиш используют для того, чтобы изменить стандартный цикл под названием GetMessage в созданной Вами функции WinMain. Этот цикл имеет вид:

```
while(GetMessage(&msg,0,0,0))
{
    TranslateMessage(&msg); /* ввод с клавиатуры */
    ...
    DispatchMessage(&msg);
}
```

С помощью функции TranslateAccelerators можно контролировать число тех сообщений, которые цикл GetMessage получает и преобразует в сообщения для меню. Когда он в ходе работы транслирует сообщения, на его выходе появляется код TRUE, а когда нет - то FALSE.

Вот как это делают обычно:

```
while(GetMessage(&msg,0,0,0))
{
    if(!TranslateAccelerators(hwnd,haccel,&msg))
    {
        TranslateMessage(&msg); /* Ввод с клавиатуры */
        DispatchMessage(&msg);
    }
}
```

Напомним, что эта функция возвращает значение FALSE, если кодовая комбинация не обнаружена, а данное сообщение обрабатывается с помощью процедур TranslateMessage и DispatchMessage. Если же кодовая комбинация обнаружена, то данное сообщение в цикле GetMessage не обрабатывается, и этим от начала до конца занимается процедура TranslateAccelerator.

Ниже приводится в законченном виде программа, в которой показано, как можно создать таблицу быстрого вызова. Она построена на основе программы Stanmenu,

создающей стандартное меню, которую мы привели в начале этой главы. Создадим четыре стандартные кодовые комбинации для работы с Clipboard соответственно для операций: Undo, Cut, Copy, Paste.

MAKEFILE.MAK

.AUTODEPEND

```
# Определения транслятора
INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;;
C:\BORLANDC\INCLUDE
CC = bcc -c -D_CLASSDLL -H -ml -WS -w -I$(INC)

# Неявные правила
.c.obj:
$(CC) {S< }
.cpp.obj:
$(CC) {S< }

# Неявные правила
Accel.exe: Accel.res Accel.def Accel.obj
tlink /c /C /n /P- /Twe /x @Accel.LNK
RC Accel.res Accel.exe

# Зависимости отдельных файлов
Accel.obj: Accel.cpp
Accel.res: Accel.rc Accel.cur Accel.ico
RC -R -FO Accel.res Accel.RC
```

ACCEL.LNK

```
c:\borlandc\lib\cOwl.obj+
Accel.obj
Accel, Accel
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwl.lib+
import.lib+
mathl.lib+
cl.lib
Accel.def
```

ACCEL.H

// Определение кодов команд

```
#define IDM_FILE_NEW      22000
#define IDM_FILE_OPEN     22001
#define IDM_FILE_SAVE     22002
#define IDM_FILE_SAVEAS   22003
#define IDM_FILE_PRINT    22004
#define IDM_FILE_EXIT     22005
#define IDM_EDIT_UNDO     22006
#define IDM_EDIT_CUT      22007
#define IDM_EDIT_COPY     22008
#define IDM_EDIT_PASTE    22009
#define IDM_EDIT_CLEAR    22010
#define IDM_EDIT_DELETE   22011
```



```
/*
ACCEL.CPP - Программа, демонстрирующая задание клавиш быстрого вызова
*/
```

```
#include <owl.h>
#include "accel.h"
```

```
#define COMMANDMSG(arg) (arg.WParam)
```

```
/*
Описания классов
*/
```

```
class TAccelApplication : public TApplication
{
public:
    TAccelApplication (LPSTR lpszName, HANDLE hInstance,
                      HANDLE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow);
    virtual void InitMainWindow ();
};
```

```
class TAccelWindow : public TWindow
{
public:
    TAccelWindow (PTWindowsObject pwParent, LPSTR lpszTitle,
                  PTModule pmModule);
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WNDCLASS&);
    virtual void WMCommand(TMessage& Msg)=[WM_COMMAND];
};
```

```
/*
Функция WinMain.
*/
```

```
int PASCAL WinMain (HANDLE hInstance,
                    HANDLE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow)
{
    TAccelApplication Accel ("Accel", hInstance, hPrevInstance,
                             lpszCmdLine, nCmdShow);
    Accel.Run();
    return Accel.Status;
}
```

```
/*
Компонентная функция класса прикладной программы
*/
```

```
TAccelApplication::TAccelApplication (LPSTR lpszName, HANDLE hInstance,
                                       HANDLE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow)
    :TApplication (lpszName, hInstance,
                  hPrevInstance, lpszCmdLine, nCmdShow)
{
    HAccTable = LoadAccelerators (hInstance, "#1");
}
```

```
/*
Компонентная функция класса прикладной программы
*/
```

```
void TAccelApplication::InitMainWindow ()
{
```



```

MainWindow = new TAccelWindow (NULL, "Accelerators", NULL);
}

/*
Компонентная функция класса TAccelWindow
*/
TAccelWindow::TAccelWindow (PTWindowsObject pwParent,
                             LPSTR lpszTitle, PTModule pmModule)
: TWindow(pwParent, lpszTitle, pmModule)
{
    /* Здесь инициализируется окно */
}

/*
Компонентная функция класса TAccelWindow
*/
LPSTR TAccelWindow::GetClassName ()
{
    return "Accel:MAIN";
}

/*
Компонентная функция класса TAccelWindow
*/
void TAccelWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
    wc.lpszMenuName = "#1";
}

/*
Компонентная функция класса TAccelWindow
*/
void TAccelWindow::WMCommand(TMessage& Msg)
{
    char buffer[80];

    if (COMMANDMSG(Msg) == IDM_FILE_EXIT)
        SendMessage (HWindow, WM_SYSCOMMAND, SC_CLOSE, 0L);
    else
    {
        wsprintf (buffer, "Command = %d", COMMANDMSG(Msg));
        MessageBox (HWindow, buffer, "WM_COMMAND", MB_OK);
    }
}

```

ACCEL.RC

```

#include <windows.h>
#include "accel.h"

snapshot icon Accel.ico
hand cursor Accel.cur
1 MENU
{
    POPUP "&File"
    {

```



```

    MENUITEM "&New",      IDM_FILE_NEW
    MENUITEM "&Open...",  IDM_FILE_OPEN
    MENUITEM "&Save",     IDM_FILE_SAVE
    MENUITEM "Save &As...", IDM_FILE_SAVEAS
    MENUITEM SEPARATOR
    MENUITEM "&Print",    IDM_FILE_PRINT
    MENUITEM SEPARATOR
    MENUITEM "E&xit",     IDM_FILE_EXIT
}
POPUP "&Edit"
{
    MENUITEM "&Undo\tAlt+Backspace", IDM_EDIT_UNDO
    MENUITEM SEPARATOR
    MENUITEM "Cu&t\tShift+Del",      IDM_EDIT_CUT
    MENUITEM "&Copy\tCtrl+Ins",      IDM_EDIT_COPY
    MENUITEM "&Paste\tShift+Ins",    IDM_EDIT_PASTE
    MENUITEM SEPARATOR
    MENUITEM "Cl&ear",               IDM_EDIT_CLEAR :
    MENUITEM "&Delete",              IDM_EDIT_DELETE :
}
}
1 ACCELERATORS
{
    VK_BACK,  IDM_EDIT_UNDO, VIRTKEY, ALT
    VK_DELETE, IDM_EDIT_CUT, VIRTKEY, SHIFT
    VK_INSERT, IDM_EDIT_COPY, VIRTKEY, CONTROL
    VK_INSERT, IDM_EDIT_PASTE, VIRTKEY, SHIFT
}

```

ACCEL.DEF

```

NAME ACCEL
EXETYPE WINDOWS

DESCRIPTION 'Accel - Sample Accelerator'
CODE MOVEABLE DISCARDABLE
DATA MOVEABLE MULTIPLE
HEAPSIZE 512
STACKSIZE 5120

```

Все, что было связано с заданием и включением в работу кодовых комбинаций клавиш в нужном нам коде, выполнялось с помощью функций конструктора объектов TAccelApplication. С их помощью мы загрузили таблицу ускоренного вызова в память.

Циклы обработки в объекте уровня приложений OWL поддерживают кодовые комбинации клавиш. В частности, в его подпрограмме TranslateAccelerator осуществляется опрос комбинаций клавиш и распределение потока поступающих сообщений по предназначению. Вы также, очевидно, заметили, что вместо использования числовых констант в ресурсном файле мы задали ряд символьных констант в присоединенном файле ACCEL.H. Это позволило нам вместо так называемых магических цифр, о назначении которых никто толком сказать ничего не может, использовать гораздо более понятную переменную IDM_NEW.

Как ресурсный, так и исходный файл языка C осуществляют доступ к переменным, которые записаны в файле ACCEL.H при использовании уже известного нам выражения:

```
#include "accel.h"
```


Среда Windows, таким образом, включает в себя все необходимые процедуры и программы управления. С ее помощью можно задавать кодовые комбинации клавиш для управления Вашей программой с помощью клавиатуры. Она может подсказать пользователю меню с нужным сообщением, как выполнить любое действие. Система ее меню гибкая и быстрая. С помощью среды Windows Вы можете создавать неподвижные шаблоны для меню, быстро строить нужные меню, вносить изменения в них.

В следующей главе мы с Вами узнаем, как с помощью машинной графики сделать наши меню красивее.

В ней мы подробно рассмотрим методы, которые позволят нам, используя графические процедуры GDI, изменять облик наших меню. Этих методов три:

- с контрольными метками или галочками, вид которых принят по умолчанию;
- с использованием растров;
- с рисунками в меню, создаваемыми автором.

Глава 12. Расширение графических возможностей меню

Для прорисовки текста внутри элементов меню Windows используется системный шрифт (типа Helvetica, жирный, пропорциональный). Этот выбранный по умолчанию шрифт используется потому, что его внешний вид современен, хорошо проработан и его легко читать. Однако могут возникнуть ситуации, когда Вы захотите использовать иной тип шрифта внутри элемента меню. У вас может возникнуть желание сконструировать меню, которое содержало бы внутри своих элементов геометрические фигуры или графические образы. Когда Вы пожелаете это сделать, Windows даст возможность "внедриться" в механизм создания меню, чтобы Вы смогли расширить графические возможности своего меню. В этой главе мы займемся рассмотрением трех методов, каждым из которых Вы сможете воспользоваться: элементы пользовательской прорисовки меню, битовые отображения внутри меню и контрольные маркеры владельца.

Элементы пользовательской прорисовки меню

Все меню, созданием которых мы занимались до сих пор, использовали для прорисовки текстовой метки системный шрифт. Но вполне вероятно, что может возникнуть необходимость заменить текст каким-либо графическим объектом либо заменить используемый шрифт другим. Элементы пользовательской прорисовки меню дают возможность создавать графические образы меню, прорисовывая их с помощью GDI. Вместо того чтобы сообщать пользователю о содержании элемента меню, Вы можете непосредственно его показать. Программа прорисовки должна использовать эту возможность, чтобы давать пользователю информацию о различных линиях, формах и шрифтах, которыми он может пользоваться. Программа регистрации потока должна выдавать в меню различные элементы регистрации потока, обеспечивая моментальную обратную связь по каждому возможному элементу регистрации потока.

Исходный компилятор не обеспечивает какой-либо поддержки элементов пользовательской прорисовки меню, поэтому Вам необходимо в динамике создавать элементы меню, обращаясь к одной из процедур модификации меню, о которых мы говорили в предыдущей главе. Если Вы хотите, чтобы элемент пользовательской прорисовки меню появлялся в меню при первом появлении окна, то вероятнее всего, Вы захотите добавить элемент меню в ответ на сообщение WM_CREATE. Вы не можете добавлять элементы пользовательской прорисовки меню в момент создания объекта окна (например, в конструкторе объекта Вашего окна) до тех пор, пока не будет создано окно Windows.

После того как элемент пользовательской прорисовки меню вставлен в меню, в процедуру окна передаются два сообщения для измерения и прорисовки элемента меню - WM_MEASUREITEM и WM_DRAWITEM. Сообщение WM_MEASUREITEM поступает до того, как элемент меню прорисован. Ваша программа должна ответить на это сообщение, заполнив структуру данных информацией, описывающей ширину и высоту Вашего элемента меню в пикселях. Второе сообщение WM_DRAWITEM посылается всегда, когда необходимо прорисовывать элемент меню, т.е. при выводе меню на экран и тогда, когда элемент меню выделяется подсветкой для визуальной обратной связи с пользователем при выборе элемента из меню.

Перед тем, как мы подробно рассмотрим обработку этих двух сообщений, будет полезно заметить, что эти два сообщения посылаются и при создании объектов пользовательской прорисовки других типов: кнопок, комбинированных блоков и списочных блоков. Как Вы позже убедитесь, существуют некоторые элементы механизма пользовательской прорисовки, использование которых предназначено для других объектов пользовательской прорисовки, их можно благополучно проигнорировать тогда, когда Вы имеете дело с элементами пользовательской прорисовки меню. Постепенно изучая этот механизм, мы обязательно укажем Вам на эти элементы.

Сообщение WM_MEASUREITEM

Элемент пользовательской прорисовки может быть большим или маленьким, таким, каким Вы хотите его сделать. Сообщение WM_MEASUREITEM передает системе точный размер, необходимый для прорисовки элемента меню. Данные выражаются в пикселях, но при вызове для прорисовки процедур GDI Вы можете использовать соответствующую GDI систему картографических координат.

При расчете размера области, необходимой для элемента пользовательской прорисовки меню, Вы должны учитывать область вокруг элемента меню, т.е. требуется достаточное пространство между элементом меню и прочими его элементами. Неплохо также зарезервировать пространство под контрольный маркер в левой части меню. Вы можете обратиться к процедуре GetMenuCheckMarkDimensions за информацией о ширине и высоте контрольного маркера по умолчанию. Но даже если Вы не собираетесь использовать контрольный маркер в элементе пользовательской прорисовки меню, все же придется это сделать. Ширина контрольного маркера является стандартной левой границей элемента меню; использование этой величины поможет Вам геометрически согласовать элементы пользовательской прорисовки меню с обычными элементами меню.

Для того чтобы приспособиться под различную разрешающую способность всевозможных терминальных устройств, полезно избежать жесткого кодирования специфических размеров элементов пользовательской прорисовки меню. Следует также сделать Вашу прорисовку пропорциональной размерам системных объектов, поскольку они автоматически масштабируются при изменении разрешающей способности устройств. Пиктограмма является типичным системным объектом, который может служить хорошей точкой отсчета для элементов пользовательской прорисовки меню. Размеры пиктограммы определяются драйвером устройства системного экрана, например программой GetSystemMetrics. Если Вы хотите сделать элементы меню того же размера, что и пиктограмма, то делайте это следующим образом:

```
DWORD dwCheckMark;  
int cxWidth, cyHeight;  
  
/* Ширина */  
cxWidth = GetSystemMetrics (SM_CXICON);  
  
/* Высота */  
cyHeight = GetSystemMetrics (SM_CYICON);  
  
/* Настройка для контрольного маркера */  
dwCheckMark = GetMenuCheckMarkDimensions();  
cxWidth += LOWORD(dwCheckMark);  
cyHeight = max(HIWORD(dwCheckMark), cyHeight);
```

Мы добавили величину для ширины контрольного маркера, а также переписали cyHeight, чтобы удостовериться в настройке высоты контрольного маркера.

В качестве альтернативы можно базировать размер Вашего элемента пользовательской прорисовки меню на метрических параметрах системного шрифта. Когда в главе 14 мы будем обсуждать диалоговые блоки, Вы увидите, что менеджер диалогового блока использует этот метод для построения аппаратно-зависимых координат диалогового блока. Для расчета габаритов элемента пользовательской прорисовки меню, который в два раза выше системного шрифта и имеет ширину, равную ширине 15 символов, Вы можете сделать следующее:

```
DWORD dwCheckMark;  
HDC hdc  
int cxWidth, cyHeight;
```



```

TEXTMETRIC tm;
hdc = GetDC(hwnd);
GetTextMetrics(hdc, &tm);
ReleaseDC(hwnd, hdc);

cxWidth = tm.tmMaxCharWigth * 15;
cyHeight = tm.tmHeight * 2;

/* Настройка для контрольного маркера */

dwCheckMark = GetMenuCheckMarkDimensions();
cxWidth += LOWORD(dwCheckMark);
cyHeight = max(HIWORD(dwCheckMark), cyHeight);

```

Процедура `GetTextMetrics`, о которой мы упоминали в главе 10, передает метрические параметры шрифта, установленного в текущий момент в контексте устройства. При отсутствии специфического стиля класса окна (описан в главе 13) `GetDC` передает контекст устройства с установленным системным шрифтом. Как и в предыдущем примере, этот код включает габариты контрольного маркера меню во все размеры элемента пользовательской прорисовки.

При любом методе одно сообщение `WM_MEASUREITEM` направляется каждому элементу пользовательской прорисовки меню. Когда оно приходит, параметр `lParam` содержит длинный указатель структуры `MEASUREITEMSTRUCT`, определенной в `WINDOWS.H` следующим образом:

```

typedef struct tagMEASUREITEMSTRUCT
{
    WORD    CtlType;      /* ODT_MENU */
    WORD    CtlID;        /* Для меню игнорируется */
    WORD    itemID;
    WORD    itemWidth;    /* Передача ширины */
    WORD    itemHeight;   /* Передача высоты */
    DWORD   itemData;
}

MEASUREITEMSTRUCT;

```

Когда Вы получаете сообщение `WM_MEASUREITEM`, Вы используете поля `itemWigth` и `itemHeight` для передачи ширины и высоты своего элемента пользовательской прорисовки. Остальные поля этой структуры данных помогают идентифицировать специфический элемент пользовательской прорисовки, поскольку есть лишь одно сообщение `WM_MEASUREITEM` для каждого имеющегося элемента пользовательской прорисовки меню. При работе с элементами пользовательской прорисовки меню одно из этих полей `CtlID` может игнорироваться, поскольку оно используется только для средств управления диалоговым блоком пользовательской прорисовки.

Поле `CtlType` описывает тип элемента пользовательской прорисовки, для меню оно устанавливается в значение `ODT_MENU`. Наличие этого поля обеспечивает поддержку объектов пользовательской прорисовки в меню, списочных и комбинированных блоках.

Поле `itemID` является результирующим командным кодом для элемента меню. Это величина, передаваемая в параметр `wParam` сообщения `WM_COMMAND`. Как мы потом убедимся, эта величина устанавливается при создании элемента меню.

Поле `itemData` представляет иной способ идентификации элемента пользовательской прорисовки меню - это 32-битовое поле, которое Вы создаете и поддерживаете при создании элемента пользовательской прорисовки меню. Когда Вы создаете всплывающее меню пользовательской прорисовки (в отличие от элемента обычного

командного меню), только это поле может различить одно всплывающее меню от другого, поскольку элементы всплывающего меню не имеют результирующего командного кода.

Если Вы создаете элемент пользовательской прорисовки меню путем вызова AppendMenu, т.е. таким образом:

```
AppendMenu(hmenu, MF_OWNERDRAW, 38, (LPSTR)200);
```

то эта процедура создает элемент меню с itemID, равным 38, и itemData, равным 200. Последний параметр классифицируется как LPSTR, поскольку именно так задумана процедура. С другой стороны, компилятор выдает избыточное сигнальное сообщение. Если переменные cxWidth и cyHeight содержат описание размеров данного элемента меню, то мы можем ответить на сообщение WM_MEASUREITEM следующим образом:

```
case WM_MEASUREITEM:
{
    LPMEASUREITEMSTRUCT lpmi;
    lpmi = (LPMEASUREITEMSTRUCT)lParam;

    if(lpmi->itemID = 38)
    {
        lpmi->itemWigth = cxWigth;
        lpmi->itemHeight = cyHeight;
    }
}
```

Другой подход предполагает контроль itemData для специфического элемента меню пользовательской прорисовки:

```
case WM_MEASUREITEM:
{
    LPMEASUREITEMSTRUCT lpmi;
    lpmi = (LPMEASUREITEMSTRUCT)lParam;

    if(lpmi->itemData = 200)
    {
        lpmi->itemWigth = cxWigth;
        lpmi->itemHeight = cyHeight;
    }
}
```

Программа лишь принимает одиночное сообщение WM_MEASUREITEM для каждого элемента пользовательской прорисовки. После этого система меню запоминает размеры элемента меню и посылает сообщение WM_DRAWITEM всегда, когда необходимо нарисовать элемент меню.

Сообщение WM_DRAWITEM

Сообщение WM_DRAWITEM направляется в процедуру Вашего окна тогда, когда открывается всплывающее меню, содержащее элемент меню пользовательской прорисовки. Затем после того, как пользователь сделает выбор из этого меню, сообщение направляется для переключения подсветки меню. Как потом выяснится, сообщение WM_DRAWITEM контролирует флаги, дающие информацию, необходимую для системы меню. Когда сообщение WM_DRAWITEM отправляется, параметр процедуры окна lParam содержит длинный указатель структуры типа DRAWITEMSTRUCT. Эта структура определена в WINDOWS.H следующим образом:


```
typedef struct tagDRAWITEMSTRUCT
{
    WORD    CtlType;        /* ODT_MENU */
    WORD    CtlID;          /* Для меню игнорируется */
    WORD    itemID;
    WORD    itemAction;     /* Для меню игнорируется */
    WORD    itemState;      /* Выбирается,запрещается, контролируется */
    HWND    hwndItem;       /* Идентификатор всплывающего меню */
    HDC      hDC;
    RECT     rcItem;        /* Ограничительная рамка */
    DWORD    itemData;
} DRAWITEMSTRUCT;
```

CtlType, CtlID, itemID и itemData - четыре элемента этой структуры данных, идентичные элементам той структуры данных, которая передается с сообщением WM_MEASUREITEM:

CtlType описывает тип элемента пользовательской прорисовки. Для меню оно устанавливается в значение ODT_MENU;

CtlID может игнорироваться для элементов пользовательской прорисовки меню, поскольку оно используется только для средств управления диалоговым блоком пользовательской прорисовки;

itemID - результирующий командный код для элемента меню. Это та величина, которую передают в параметр wParam сообщения WM_COMMAND;

itemData представляет иной способ идентификации элемента пользовательской прорисовки меню и единственную возможность различить одно всплывающее меню пользовательской прорисовки от другого.

Поле itemAction описывает то действие, которое с точки зрения системы меню Вы должны выполнить. Однако, если элементы меню так нарисованы, что отражают состояние элемента меню, Вы можете игнорировать это поле и определять состояние меню по следующему полю.

Поле itemState определяет текущее состояние элемента меню, как показано ниже:

Значение	Состояние элемента меню
ODS_CHECKED	В меню должен появиться контрольный маркер
ODS_DISABLED	Меню, которое должно быть прорисовано, отключается
ODS_FOCUS	Игнорируется, не используется для меню
ODS_GRAYED	Меню, которое должно быть прорисовано, запрещается
ODS_SELECTED	Меню, которое должно быть прорисовано, выбирается

Наиболее важен флаг ODS_SELECTED. Когда этот бит установлен, данный элемент меню должен быть высвечен. Высвеченный элемент меню прорисовывается иным цветом по сравнению со всеми остальными элементами меню, чтобы обеспечить визуальную обратную связь для пользователя, работающего с меню. Вы должны прорисовывать высвеченные элементы меню, используя два системных цвета, которые задаются для этой цели. Для получения величин RGB требуемых системных цветов следует вызвать процедуру GetSysColor. Далее приведены индексы цветов, которые Вы должны использовать для подсветки меню:

Значение	Описание
COLOR_HIGHLIGHT	Цвет фона объекта подсветки
COLOR_HIGHLIGHTTEXT	Основной цвет объекта подсветки

Если Вы планируете установить контроль за состоянием элемента пользовательской прорисовки меню или запретить элемент меню, то Ваш ответ на сообщение WM_DRAWITEM должен это гарантировать. Необходимо проверить величину в по-

ле `itemState` и изменить способ, которым Вы прорисовываете элемент Вашего меню, в соответствии с желаемым результатом. Например, Вы должны использовать оттенок серым цветом в своей прорисовке, если хотите отметить запрещенный элемент меню. Вы можете нарисовать контрольный маркер, используя битовое отображение, встроенное в драйвер экрана. Здесь показан один из способов получения идентификатора для этого битового отображения и прорисовки с ним:

```
hbm          = LoadBitmap(NULL, MAKEINTRESOURCE(OBM_CHECK));
hdcBitmap    = CreateCompatibleDC(hdc);
hbmOld       = SelectObject(hdcBitmap, hbm);
dwDimensions = GetMenuCheckMarkDimensions();

cxWigth      = LOWORD(dwDimensions);
cyHeight     = HIWORD(dwDimensions);

BitBlt(ps.hdc, 0, 0, cxWigth, cyHeight, hdcBitmap, 0, 0, SRCCOPY);
SelectObject(hdcBitmap, hbmOld);
DeleteDC(hdcBitmap);
```

Несмотря на свое название, поле `hwndItem` не является идентификатором окна, наоборот, оно является идентификатором всплывающего меню для меню, содержащего элемент пользовательской прорисовки. Его название оправдано совместным использованием со средствами управления диалогового блока пользовательской прорисовки, когда оно содержит идентификатор окна. Когда программа принимает сообщение `WM_DRAWITEM`, она использует идентификатор всплывающего меню для вызова процедуры запроса меню для получения, например, данных о количестве элементов, входящих в текущее меню, или о состоянии остальных элементов меню.

Поле `hDC` идентифицирует контекст устройства для прорисовки в меню. После обработки убедитесь в том, что контекст устройства установлен в исходное состояние, в противном случае Вы создадите проблемы для системы меню при прорисовке остальных частей меню.

Поле `rcItem` задает прямоугольник, внутри которого Вы можете прорисовывать элемент меню. Вам не следует выходить за рамки этого прямоугольника, в противном случае Вы рискуете закрасить соседние элементы меню.

Элементы пользовательской прорисовки, использующие эти сообщения и структуры данных, предоставляют программе достаточно гибкие возможности для определения того, каким образом должен появиться на экране элемент меню. Давайте рассмотрим пример программы, в которой показано, каким образом следует использовать в своей работе описанные здесь сообщения и структуры данных.

Пример программы: Owndraw

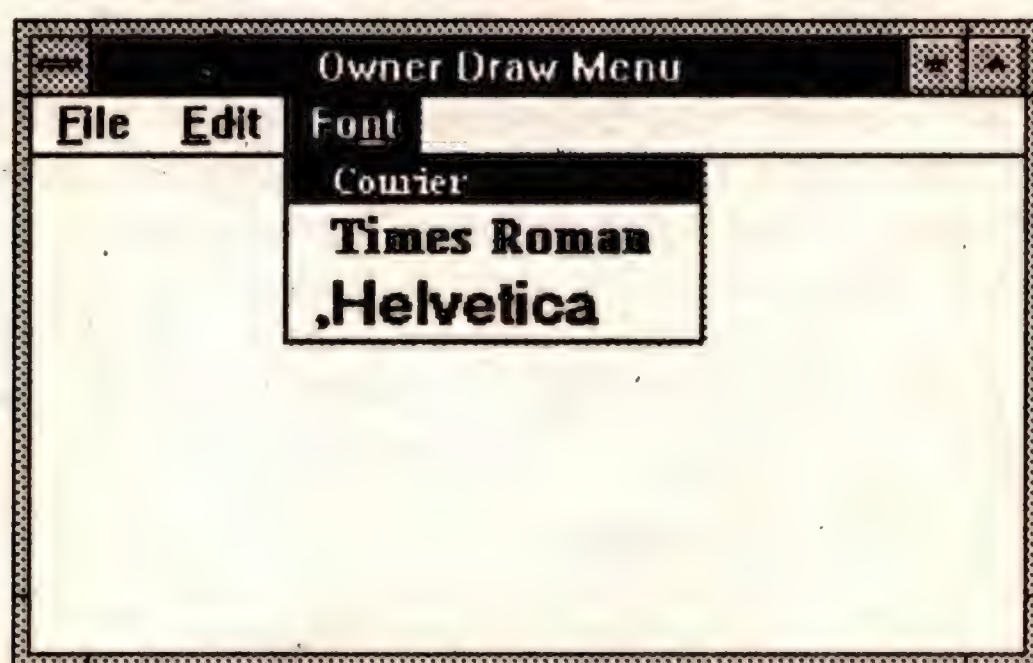


Рис.12.1. Меню пользовательской прорисовки, демонстрирующее три шрифта.

Наш пример программы `Owndraw` (рис.12.1) демонстрирует, как элементы пользовательской прорисовки меню используются для вывода на экранное меню различ-

ных шрифтов. Пользователь задает не только название шрифта, но и саму форму шрифта. Эта технология полезна в любой программе, позволяющей пользователю выбирать шрифт. Если Вы поступите так, то обязательно отметите, что для прорисовки меню можно использовать только экранные шрифты. Следовательно, если Вы захотите продемонстрировать пользователю допустимые шрифты принтера, то иногда Вас будут ограничивать названия шрифтов.

Далее приводится исходный текст программы Owndraw. Для написания этой программы мы использовали программу Stanmenu в качестве базовой, прибавили к ней элементы, необходимые для наших элементов пользовательской прорисовки.

MAKEFILE.MAK

.AUTODEPEND

Определения транслятора

INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;
C:\BORLANDC\INCLUDE

CC =bcc -c -D_CLASSDLL -H -ml -WS -w -I\$(INC)

Неявные правила

c.obj:

\$(CC) {\$<}

.cpp.obj:

\$(CC) {\$<}

Явные правила

OwnDraw.exe: OwnDraw.res OwnDraw.def OwnDraw.obj

tlink /c/C/n/P-/Twe/x @OwnDraw.LNK

RC OwnDraw.res OwnDraw.exe

Зависимости отдельных файлов

OwnDraw.obj: OwnDraw.cpp

OwnDraw.res: OwnDraw.rc OwnDraw.cur OwnDraw.ico

RC -R -FO OwnDraw.res OwnDraw.RC

OWNDRAW.LNK

c:\borlandc\lib\cOwl.obj+

OwnDraw.obj

OwnDraw,OwnDraw

\borlandc\owl\lib\owl.lib+

crtll.lib+

cwl.lib+

import.lib+

mathl.lib+

cl.lib

OwnDraw.

OWNDRAW.CPP

/*

OWNDRAW.CPP - Пример собственного меню

*/

#include <owl.h>

#include "owndraw.h"

#define max(a,b) (((a) > (b)) ? (a) : (b))

#define COMMANDMSG(arg) (arg.WParam)

/*

Описания классов

*/

```
class TOwnDrawApplication : public TApplication
{
public:
    TOwnDrawApplication (LPSTR lpszName,
                        HANDLE hInstance,
                        HANDLE hPrevInstance,
                        LPSTR lpszCmdLine,
                        int nCmdShow);

    virtual void InitMainWindow ();
};

class TOwnDrawWindow : public TWindow
{
public:
    TOwnDrawWindow (PTWindowsObject pwParent,
                    LPSTR lpszTitle,
                    PTModule pmModule);

    ~TOwnDrawWindow();
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WNDCLASS&);
    virtual void WMCommand(TMessage& Msg) = [WM_COMMAND];
    virtual void WMCreate(TMessage& Msg) = [WM_CREATE];
    virtual void WMDrawItem(TMessage& Msg) = [WM_DRAWITEM];
    virtual void WMMeasureItem (TMessage& Msg) = [WM_MEASUREITEM];

private:
    HFONT hfontTmsRmn;
    HFONT hfontHelv;
    HFONT hfontCour;
    COLORREF crHighlight;
    COLORREF crHighlightText;
};
```

/*

Функция WinMain

*/

```
int PASCAL WinMain(HANDLE hInstance,
                  HANDLE hPrevInstance,
                  LPSTR lpszCmdLine,
                  int nCmdShow)
{
    TOwnDrawApplication OwnDraw ("OwnDraw",
                                hInstance,
                                hPrevInstance,
                                lpszCmdLine,
                                nCmdShow);

    OwnDraw.Run();
    return OwnDraw.Status;
}
```



```

/*
Компонентная функция класса прикладной программы
*/

```

```

TOwnDrawApplication:: TOwnDrawApplication
    (LPSTR lpszName,
     HANDLE hInstance,
     HANDLE hPrevInstance,
     LPSTR lpszCmdLine,
     int nCmdShow )

```

```

    :TApplication(lpszName,
                  hInstance,
                  hPrevInstance,
                  lpszCmdLine,
                  nCmdShow)

```

```

{
    /* Здесь инициализируется прикладная программа. */
}

```

```

/*
Компонентная функция класса прикладной программы
*/

```

```

void TOwnDrawApplication:: InitMainWindow ()
{
    MainWindow = new TOwnDrawWindow (NULL,
                                      "Owner Draw Menu",
                                      NULL);
}

```

```

/*
Компонентная функция класса TOwnDrawWindow
*/

```

```

TOwnDrawWindow::TOwnDrawWindow (PTWindowsObject pwParent,
                                  LPSTR lpszTitle,
                                  PTModule pmModule)

```

```

    :TWindow (pwParent, lpszTitle, pmModule)

```

```

{
    HDC          hdc;
    LOGFONT      lf;
    TEXTMETRIC    tm;

```

```

/*
Создание трех логических шрифтов с различными размерами и стилями sizes
and styles
*/

```

```

    memset (&lf, 0, sizeof (LOGFONT));

```

```

    hdc = CreateDC ("DISPLAY", 0, 0, 0);
    GetTextMetrics (hdc, &tm);
    DeleteDC (hdc);

```

```

    // Шрифт 1: Bold Courier.

```

```

    lf.lfWeight = 700;
    lstrcpy ((LPSTR)&lf.lfFaceName[0], "Courier");
    lf.lfHeight = tm.tmHeight;
    hfontCour = CreateFontIndirect (&lf);

```


// Шрифт 2: Times Roman.

```
lstrcpy ((LPSTR)&lf.lfFaceName[0], "Tms Rmn");  
lf.lfHeight = tm.tmHeight + tm.tmHeight / 4;  
hfontTmsRmn = CreateFontIndirect (&lf);
```

// Шрифт 3: Helv.

```
lstrcpy ((LPSTR)&lf.lfFaceName[0], "Helv");  
lf.lfHeight = tm.tmHeight + tm.tmHeight / 2;  
hfontHelv = CreateFontIndirect (&lf);
```

/*

Выбор системных цветов для прорисовки пунктов меню

*/

```
crHighlight = GetSysColor (COLOR_HIGHLIGHT);  
crHighlightText = GetSysColor (COLOR_HIGHLIGHTTEXT);  
}
```

/*

Компонентная функция класса TOwnDrawWindow

*/

TOwnDrawWindow::~TOwnDrawWindow ()

```
{  
    DeleteObject (hfontTmsRmn);  
    DeleteObject (hfontHelv);  
    DeleteObject (hfontCour);  
}
```

/*

Компонентная функция класса TOwnDrawWindow

*/

LPSTR TOwnDrawWindow::GetClassName ()

```
{  
    return "OwnDraw:MAIN";  
}
```

/*

Компонентная функция класса TOwnDrawWindow

*/

void TOwnDrawWindow::GetWindowClass (WNDCLASS& wc)

```
{  
    TWindow::GetWindowClass (wc);  
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");  
    wc.hCursor=LoadCursor (wc.hInstance, "hand");  
    wc.lpszMenuName = "#1";  
}
```

/*

Компонентная функция класса TOwnDrawWindow

*/

void TOwnDrawWindow::WMCommand(TMessage& Msg)

```
{  
    char buffer[80];  
    if (COMMANDMSG(Msg) == IDM_FILE_EXIT)  
        SendMessage (HWindow, WM_SYSCOMMAND, SC_CLOSE, 0L);  
}
```



```

else
{
    wsprintf(buffer, "Command = %d", COMMANDMSG(Msg));
    MessageBox(HWindow, buffer, "WM_COMMAND", MB_OK);
}
}

/*
Компонентная функция класса TOwnDrawWindow
*/

void TOwnDrawWindow::WMCreate(TMessage& Msg)
{
    HMENU hmenu;
    HMENU hmenuPopup;

    /*
    Создание трех собственных пунктов меню
    */

    hmenu = GetMenu (Msg.Receiver);
    hmenuPopup = CreatePopupMenu();
    AppendMenu (hmenu, MF_POPUP, hmenuPopup, "Fo&nt");
    AppendMenu (hmenuPopup,
        MF_OWNERDRAW,
        IDM_FONT_COURIER,
        MAKEINTRESOURCE(IDM_FONT_COURIER));

    AppendMenu (hmenuPopup,
        MF_OWNERDRAW,
        IDM_FONT_TMSRMN,
        MAKEINTRESOURCE(IDM_FONT_TMSRMN));

    AppendMenu (hmenuPopup,
        MF_OWNERDRAW,
        IDM_FONT_HELV,
        MAKEINTRESOURCE(IDM_FONT_HELV));
}

/*
Компонентная функция класса TOwnDrawWindow
*/

void TOwnDrawWindow::WMDrawItem(TMessage& Msg)
{
    char buff[80];
    COLORREF crFore;
    COLORREF crBack;
    DWORD dwCheckMark;
    HFONT hfontOld;
    HFONT hfont;
    int fSwapColors;
    int xText, yText;
    int cxCheck;
    LPDRAWITEMSTRUCT lpdi;

    lpdi = (LPDRAWITEMSTRUCT)Msg.LParam;
    switch (lpdi->itemID)
    {

```



```

        case IDM_FONT_COURIER:
            lstrcpy (buff, "Courier");
            hfont = hfontCour;
            break;
        case IDM_FONT_TMSRMN:
            lstrcpy (buff, "Times Roman");
            hfont = hfontTmsRmn;
            break;
        case IDM_FONT_HELV:
            lstrcpy (buff, "Helvetica");
            hfont = hfontHelv;
            break;
    }
    hfontOld = SelectObject (lpdi->hDC, hfont);

    /*
    Если текущий пункт меню должен быть подсвечен, установить правильные
    системные цвета
    */

    if (lpdi->itemState & ODS_SELECTED)
    {
        crFore = SetTextColor (lpdi->hDC, crHighlightText);
        crBack = SetBkColor (lpdi->hDC, crHighlight);
        fSwapColors = TRUE;
    }
    else
        fSwapColors = FALSE;

    /*
    Взять ширину контрольных маркеров
    */

    dwCheckMark = GetMenuCheckMarkDimensions();
    cxCheck = LOWORD (dwCheckMark);

    xText = lpdi->rcItem.left + cxCheck;
    yText = lpdi->rcItem.top;

    /*
    Вывод строки текста
    */

    ExtTextOut (lpdi->hDC,
                xText,
                yText,
                ETO_OPAQUE,
                &lpdi->rcItem,
                buff,
                lstrlen(buff),
                NULL);

    /*
    Восстановить контекст устройства перед тем, как послать его обратно
    */

    // Восстановить шрифт
    SelectObject (lpdi->hDC, hfontOld);
    if (fSwapColors)

```



```

    {
        // Восстановить цвета
        SetTextColor (lpdi->hDC, crFore);
        SetBkColor (lpdi->hDC, crBack);
    }
}

/*
Компонентная функция класса TOwnDrawWindow
*/

void TOwnDrawWindow::WMMeasureItem(TMessage& Msg)
{
    char buff[80];
    DWORD dwSize;
    DWORD dwCheckMark;
    HDC hdc;
    HFONT hfont;
    int cxWidth, cyHeight;
    int cxCheck, cyCheck;
    LPMEASUREITEMSTRUCT lpmi;

    lpmi = (LPMEASUREITEMSTRUCT)Msg.LParam;
    switch (lpmi->itemID)
    {
        case IDM_FONT_COURIER:
            lstrcpy (buff, "Courier");
            hfont = hfontCour;
            break;
        case IDM_FONT_TMSRMN:
            lstrcpy (buff, "Times Roman");
            hfont = hfontTmsRmn;
            break;
        case IDM_FONT_HELV:
            lstrcpy (buff, "Helvetica");
            hfont = hfontHelv; break;
    }

    /*
Нахождение размера строки
*/

    hdc = GetDC (HWindow);
    SelectObject (hdc, hfont);
    dwSize = GetTextExtent (hdc, buff, lstrlen(buff));
    ReleaseDC (HWindow, hdc);

    /*
Начать с информации о размере строки
*/

    cxWidth = LOWORD(dwSize);
    cyHeight = HIWORD(dwSize);

    /*
Добавить информацию о размере контрольных маркеров
*/

    dwCheckMark = GetMenuCheckMarkDimensions();

```



```

cxCheck = LOWORD (dwCheckMark);
cyCheck = HIWORD (dwCheckMark);

lpmi->itemWidth = cxWidth + cxCheck;
lpmi->itemHeight = max (cyHeight, cyCheck);
}

```

OWNDRAW.H

```

/*
OwnDraw.H - Подключаемый файл для OWNDRAW.CPP
*/

```

```

#define IDM_FILE_NEW      100
#define IDM_FILE_OPEN     101
#define IDM_FILE_SAVE     102
#define IDM_FILE_SAVEAS   103
#define IDM_FILE_PRINT    104
#define IDM_FILE_EXIT     105

#define IDM_EDIT_UNDO     200
#define IDM_EDIT_CUT      201
#define IDM_EDIT_COPY     202
#define IDM_EDIT_PASTE    203
#define IDM_EDIT_CLEAR    204
#define IDM_EDIT_DELETE   205

#define IDM_FONT_TMSRMN   300
#define IDM_FONT_HELV     301
#define IDM_FONT_COURIER  302

```

OWNDRAW.RC

```

#include "owndraw.h"

snapshot icon owndraw.ico

hand cursor owndraw.cur

1 MENU
{
    POPUP "&File"
    {
        MENUITEM "&New",          IDM_FILE_NEW
        MENUITEM "&Open...",      IDM_FILE_OPEN
        MENUITEM "&Save",          IDM_FILE_SAVE
        MENUITEM "Save &As...",    IDM_FILE_SAVEAS
        MENUITEM SEPARATOR
        MENUITEM "&Print",         IDM_FILE_PRINT
        MENUITEM SEPARATOR
        MENUITEM "E&xit",          IDM_FILE_EXIT
    }

    POPUP "&Edit"
    {
        MENUITEM "&Undo\tAlt+Backspace", IDM_EDIT_UNDO
        MENUITEM SEPARATOR
        MENUITEM "Cu&t\tShift+Del",      IDM_EDIT_CUT
        MENUITEM "&Copy\tCtrl+Ins",       IDM_EDIT_COPY
        MENUITEM "&Paste\tShift+Ins",     IDM_EDIT_PASTE
    }
}

```



```

MENUITEM SEPARATOR
MENUITEM "Cl&ear",
MENUITEM "&Delete",
}

```

```

IDM_EDIT_CLEAR
IDM_EDIT_DELETE

```

OWNDRAW.DEF

NAME OWNDRAW

EXETYPE WINDOWS

DESCRIPTION 'OwnDraw - Owner draw menus'

CODE MOVEABLE DISCARDABLE

DATA MOVEABLE MULTIPLE

HEAPSIZE 512

STACKSIZE 5120

Подобно программам, рассмотренным ранее, эта программа имеет два объекта: прикладной и окно. Мы достаточно часто использовали прикладной объект OWL как он есть. Этот объект выполняет всю необходимую инициализацию и обслуживает цикл нашего сообщения.

Как мы уже убедились, конструктор объекта окна инициализирует различные элементы данных этого объекта. Этот процесс включает создание трех шрифтов GDI и восстановление информации о цвете, которую будет использовать объект окна при прорисовке своих элементов меню пользовательской прорисовки. Хотя реальная мировая прикладная программа больше подходила бы для создания объекта меню пользовательской прорисовки, мы несколько упрощаем их ради цели данной книги - помочь Вам разобраться в принципе работы Windows.

В ответ на сообщение WM_CREATE программа Owndraw создает три элемента меню пользовательской прорисовки и передает их в окно. Вы не можете добавить меню в окно через конструктор объекта окна до тех пор, пока не будет создано окно Windows. В отличие от Owndraw Вам следует передавать сообщение WM_CREATE для инициализации элемента меню пользовательской прорисовки. Фактически то же самое верно и для любого другого объекта, который Вы желаете добавить к окну.

Как только обеспечивается информация об измерениях, процедура нашего окна начинает принимать сообщения WM_DRAWITEM. Поскольку нам не интересно запрещать или отключать элементы меню, программа Owndraw не будет контролировать соответствующие флаги. Мы также не станем отслеживать, должен ли элемент меню выводить на экран контрольный маркер. Если нужен контрольный маркер, мы должны сами его нарисовать в ответ на сообщение WM_DRAWITEM. Необходимо сказать, что если мы собрались рисовать часть элемента меню, то следует нарисовать этот элемент целиком.

Мнемонические коды клавиатуры не поддерживаются элементами меню пользовательской прорисовки. И это весьма существенно, поскольку для графического образа невероятно сложно подобрать идентичную последовательность клавишных команд. Однако наличие текста внутри элементов меню пользовательской прорисовки приводит к парадоксу: хотя для элементов пользовательской прорисовки можно использовать мнемонические клавишные команды в качестве графических объектов, Windows не обеспечивает поддержку мнемонических клавишных команд для элементов пользовательской прорисовки. Если это важно для Вас, то существует способ эмуляции мнемонических кодов для элементов меню пользовательской прорисовки. Вы начинаете с подчеркивания мнемонической буквы в каждом наименовании шрифта. Затем, в ответ на сообщение WM_MENUCHAR, Вы будете контролировать мнемонические буквы и констатировать ошибку или выбор меню по переданному значению. Сообщение WM_MENUCHAR посылается тогда, когда пользователь дает последовательность клавишных команд, не соответствующую ни одной мнемонической или акселераторной клавише. Когда процедура окна по умолчанию принимает такое сообщение,

она определяет передаваемое значение, сообщаящее системе меню, что она должна дать звуковой сигнал пользователю. Но Ваша персональная обработка этого сообщения позволит обеспечить передачу необходимого значения таким образом, что система меню будет обрабатывать клавишную команду как обычный мнемонический код. Объект Вашего окна сам создает в своем конструкторе три шрифта, поэтому он должен суметь их потом и ликвидировать.

В итоге получим объекты прорисовки GDI, которые требуют своего занесения в память локальной кучи GDI. Это хороший пример того, как C++ действительно полезен программистам Windows. В любом случае всегда, когда Вы создаете в конструкторе объект данных MS Windows, убедитесь в том, что Вы ликвидировали объект в деструкторе. Указанный далее код ликвидирует шрифты в деструкторе класса объекта окна программы `Owndraw`:

```
DeleteObject(hfontTmsRmn);  
DeleteObject(hfontHelv);  
DeleteObject(hfontCour);
```

Использование элементов пользовательской прорисовки дает программе большие преимущества по управлению и делает более гибкими возможности представления элементов меню. С течением времени элемент меню может изменяться в ответ на какое-либо внешнее событие. Но Вам может и не понадобиться такая высокая гибкость в тех графических образах, которые помещены в меню. Вы имеете и другой вариант, более простой для реализации, чем элементы меню пользовательской прорисовки, подразумевающий создание битовых образов. Подобный битовый образ связан с элементом меню и автоматически выводится на экран системой меню без какого бы то ни было вмешательства с Вашей стороны.

Давайте разберемся, что представляет собой процесс помещения битового образа в элемент меню.

Битовые образы в элементах меню

В главе 6, во введении в GDI, мы обсуждали, как битовые образы используются для сохранения графических образов. Сохраненный однажды в битовом образе, графический образ может добавляться в меню вместо текста, который обычно используется при выводе меню на экран. Битовые образы в меню обеспечивают более простой по сравнению с элементами меню пользовательской прорисовки способ размещения графических образов в элементах меню. Для того чтобы в совершенстве овладеть этими возможностями, нам необходимо начать с описания, каким образом система Windows осуществляет создание и использование битовых образов GDI. Проще всего разобраться в том, что нас ожидает при работе с GDI, можно, проверив фрагмент кода, который покажет, как отдельные части объединяются в единое целое. Это один из способов создания битового образа GDI, который выводится на экранное меню:

```
HDC      hdcScreen;  
HDC      hdcBitmap;  
HBITMAP  hbm;  
HBITMAP  hbmOld;  
  
hdcScreen = GetDC(hwnd);  
hdcBitmap = CreateCompatibleDC(hdcScreen);  
hbm       = CreateCompatibleBitmap(hdc,  
                                   cxWigth,  
                                   cyHeight);  
  
ReleaseDC(hwnd, hdcScreen);  
  
hbmOld = SelectObject(hdcBitmap, hbm);
```


Использование битового образа требует, чтобы два объекта, битовый образ и контекст устройства, были созданы и связаны друг с другом. При создании битового образа нам следует указать размер прямоугольного экранного образа, который мы собираемся сохранить в памяти. GDI отреагирует выделением необходимой памяти. Контекст устройства нам необходим, поскольку, как Вы знаете, каждая процедура прорисовки GDI использует в качестве первого параметра идентификатор контекста устройства. Без такого специального "пропуска" никакой битовый образ не будет допущен в те "частные владения", к которым относятся все устройства GDI. Но имея контекст устройства, битовый образ становится полноправным пайщиком в тех возможностях прорисовки, которые предоставляет GDI.

Самый быстрый и простой способ создания контекста устройства заключается в обращении к GDI с просьбой модифицировать существующий контекст устройства путем обращения к `CreateCompatibleDC`. Эта процедура извлекает единственный параметр - идентификатор существующего контекста устройства. После того как наш битовый образ начинает копироваться на экран дисплея, мы передаем дисплейному контексту устройства то, что получили по запросу `GetDC`. Для создания битового образа, совместимого с экраном дисплея, мы обращаемся к реляционной процедуре `CreateCompatibleBitmap`. Эта процедура также использует контекст устройства как параметр, наряду со значениями ширины и высоты для определения размеров создаваемой поверхности прорисовки.

Единственное, что мы упустили, это способ соединения битового образа и контекста устройства. Процедура, которую мы используем, уже знакома, мы использовали ее для установки перьев, типов закрашки и шрифтов в контекстах устройств:

```
hbmOld = SelectObject(hdcBitmap, hbm);
```

Здесь, как Вы видите, она используется с иной целью: для установки взаимосвязи между битовым образом и контекстом устройства. После того как эта взаимосвязь установлена, мы можем применять любую процедуру прорисовки GDI для нашего битового образа. Например, мы можем обратиться к любой другой процедуре прорисовки GDI или нарисовать эллипс в виде битового образа таким способом:

```
Ellipse(hdcBitmap, x1, y1, x2, y2);
```

Закончив прорисовку в битовом образе, Вы помещаете его в меню, обращаясь к одной из следующих процедур: `AppendMenu`, `InsertMenu` или `ModifyMenu`. Здесь мы использовали `AppendMenu`, которая вставляет битовый образ в конце всплывающего меню и присваивает новому элементу меню результирующий командный код 23:

```
HMENU hmenu;  
HMENU hmenuPopup;  
  
hmenu = GetMenu(hwnd);  
hmenuPopup = GetSubMenu(hmenu, 1);  
  
AppendMenu(hmenuPopup, MF_BITMAP, 23, MAKEINTRESOURCE(hbm));
```

Когда битовый образ установлен в меню, контекст устройства больше не нужен, поэтому мы его ликвидируем. Но прежде нам необходимо извлечь из него идентификатор битового образа. Когда мы устанавливали идентификатор битового образа в наш контекст устройства, `SelectObject` передал значение, которое мы сохранили в `hbmOld`, - фиксатор для битового образа. Перед разрушением контекста устройства, мы должны прежде всего разрушить связь между контекстом устройства и битовым образом. Это выполняется следующим обращением к `SelectObject`:

```
SelectObject(hdcBitmap, hbmOld);
```

После этого мы можем приступить к разрушению контекста устройства:

```
DeleteDC(hdcBitmap);
```


На рис.12.2 показано меню, которое создано передачей пяти битовых образов в пять различных элементов меню. Каждый элемент меню представляет различный тип закрашки, который пользователь может выбрать для рисования. Это меню создано с помощью программы Bitmenu, приведенной в следующем примере.



Рис.12.2. Меню, содержащее битовые отображения с образцами закрашки GDI.

MAKEFILE.MAK

.AUTODEPEND

Определения транслятора

INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;
C:\BORLANDC\INCLUDE

CC =bcc -c -D_CLASSDLL -H -ml -WS -w -I\$(INC)

Неявные правила

.c.obj:
\$(CC) {\$<}

.cpp.obj:
\$(CC) {\$<}

Явные правила

BitMenu.exe: BitMenu.res BitMenu.def BitMenu.obj
tlink /c/C/n/P-/Twe/x @BitMenu.LNK
RC BitMenu.res BitMenu.exe

Зависимости отдельных файлов

BitMenu.obj: BitMenu.cpp

BitMenu.res: BitMenu.rc BitMenu.cur BitMenu.ico
RC -R -FO BitMenu.res BitMenu.RC

BITMENU.LNK

c:\borlandc\lib\cOwl.obj+
BitMenu.obj
BitMenu, BitMenu
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwl.lib+


```
import.lib+
mathl.lib+
cl.lib
BitMenu.def
```

BITMENU.CPP

```
/*
OWNDRAW.CPP - Пример собственного меню
*/

#include <owl.h>
#include "owndraw.h"

#define max(a,b)      (((a) > (b)) ? (a) : (b))
#define COMMANDMSG(arg) (arg.WParam)

/*
Описания классов
*/

class TOwnDrawApplication : public TApplication
{
public:
    TOwnDrawApplication (LPSTR lpszName,
                        HANDLE hInstance,
                        HANDLE hPrevInstance,
                        LPSTR lpszCmdLine,
                        int nCmdShow);

    virtual void InitMainWindow ();
};

class TOwnDrawWindow : public TWindow
{
public:
    TOwnDrawWindow (PTWindowsObject pwParent,
                    LPSTR lpszTitle,
                    PTModule pmModule);

    ~TOwnDrawWindow();
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WNDCLASS&);
    virtual void WMCommand(TMessage& Msg) = [WM_COMMAND];
    virtual void WMCreate(TMessage& Msg) = [WM_CREATE];
    virtual void WMDrawItem(TMessage& Msg) = [WM_DRAWITEM];
    virtual void WMMeasureItem (TMessage& Msg) = [WM_MEASUREITEM];

private:
    HFONT hfontTmsRmn;
    HFONT hfontHelv;
    HFONT hfontCour;
    COLORREF crHighlight;
    COLORREF crHighlightText;
};
```



```

/*
Функция WinMain
*/

```

```

int PASCAL WinMain(HANDLE hInstance,
                  HANDLE hPrevInstance,
                  LPSTR lpszCmdLine,
                  int nCmdShow)

{
    TOwnDrawApplication OwnDraw ("OwnDraw",
                                  hInstance,
                                  hPrevInstance,
                                  lpszCmdLine,
                                  nCmdShow);

    OwnDraw.Run();
    return OwnDraw.Status;
}

```

```

/*
Компонентная функция класса прикладной программы:
*/

```

```

TOwnDrawApplication::TOwnDrawApplication(LPSTR lpszName,
                                          HANDLE hInstance,
                                          HANDLE hPrevInstance,
                                          LPSTR lpszCmdLine,
                                          int nCmdShow )
    :TApplication(lpszName,
                  hInstance,
                  hPrevInstance,

```

```

                  lpszCmdLine,
                  nCmdShow)
{

```

```

    /* Здесь инициализируется прикладная программа. */
}

```

```

/*
Компонентная функция класса прикладной программы
*/

```

```

void TOwnDrawApplication::InitMainWindow ()
{
    MainWindow = new TOwnDrawWindow (NULL,
                                     "Owner Draw Menu",
                                     NULL);
}

```

```

/*
Компонентная функция класса TOwnDrawWindow
*/

```

```

TOwnDrawWindow::TOwnDrawWindow (PTWindowsObject pwParent,
                                 LPSTR lpszTitle,
                                 PTModule pmModule)
    :TWindow (pwParent,
              lpszTitle,
              pmModule)
{

```



```

HDC      hdc;
LOGFONT  lf;
TEXTMETRIC tm;

/*
Создание трех логических шрифтов с различными размерами и стилями
sizes and styles
*/

memset (&lf, 0, sizeof (LOGFONT));

hdc = CreateDC ("DISPLAY", 0, 0, 0);
GetTextMetrics (hdc, &tm);
DeleteDC (hdc);

// Шрифт 1: Bold Courier.
lf.lfWeight = 700;
lstrcpy ((LPSTR)&lf.lfFaceName[0], "Courier");
lf.lfHeight = tm.tmHeight;
hfontCour = CreateFontIndirect (&lf);

// Шрифт 2: Times Roman.
lstrcpy ((LPSTR)&lf.lfFaceName[0], "Tms Rmn");
lf.lfHeight = tm.tmHeight + tm.tmHeight/4;
hfontTmsRmn = CreateFontIndirect (&lf);

// Шрифт 3: Helv.
lstrcpy ((LPSTR)&lf.lfFaceName[0], "Helv");
lf.lfHeight = tm.tmHeight + tm.tmHeight/2;
hfontHelv = CreateFontIndirect (&lf);

/*
Выбор системных цветов для прорисовки пунктов меню
*/

crHighlight = GetSysColor (COLOR_HIGHLIGHT);
crHighlightText = GetSysColor (COLOR_HIGHLIGHTTEXT);
}

/*
Компонентная функция класса TOwnDrawWindow
*/
TOwnDrawWindow::~TOwnDrawWindow ()
{
    DeleteObject (hfontTmsRmn);
    DeleteObject (hfontHelv);
    DeleteObject (hfontCour);
}

/*
Компонентная функция класса TOwnDrawWindow
*/
LPSTR TOwnDrawWindow::GetClassName ()
{
    return "OwnDraw:MAIN";
}

```



```

/*
Компонентная функция класса TOwnDrawWindow
*/

```

```

void TOwnDrawWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
    wc.lpszMenuName = "#1";
}

```

```

/*
Компонентная функция класса TOwnDrawWindow
*/

```

```

void TOwnDrawWindow::WMCommand(TMessage& Msg)
{
    char buffer[80];
    if (COMMANDMSG(Msg) == IDM_FILE_EXIT)
        SendMessage (HWindow, WM_SYSCOMMAND, SC_CLOSE, 0L);
    else
    {
        wsprintf(buffer, "Command = %d", COMMANDMSG(Msg));
        MessageBox(HWindow, buffer, "WM_COMMAND", MB_OK);
    }
}

```

```

/*
Компонентная функция класса TOwnDrawWindow
*/

```

```

void TOwnDrawWindow::WMCreate(TMessage& Msg)
{
    HMENU    hmenu;
    HMENU    hmenuPopup;

    /*
    Создание трех собственных пунктов меню
    */

    hmenu = GetMenu (Msg.Receiver);
    hmenuPopup = CreatePopupMenu();
    AppendMenu (hmenu, MF_POPUP, hmenuPopup, "Fo&nt");
    AppendMenu (hmenuPopup,
                MF_OWNERDRAW,
                IDM_FONT_COURIER,
                MAKEINTRESOURCE(IDM_FONT_COURIER));
    AppendMenu (hmenuPopup,
                MF_OWNERDRAW,
                IDM_FONT_TMSRMN,
                MAKEINTRESOURCE(IDM_FONT_TMSRMN));
    AppendMenu (hmenuPopup,
                MF_OWNERDRAW,
                IDM_FONT_HELV,
                MAKEINTRESOURCE(IDM_FONT_HELV));
}

```



```

/*
Компонентная функция класса TOwnDrawWindow
*/
void TOwnDrawWindow::WMDrawItem(TMessage& Msg)
{
    char buff[80];
    COLORREF crFore;
    COLORREF crBack;
    DWORD dwCheckMark;
    HFONT hfontOld;
    HFONT hfont;
    int fSwapColors;
    int xText,yText;
    int cxCheck;
    LPDRAWITEMSTRUCT lpdi;

    lpdi = (LPDRAWITEMSTRUCT)Msg.LParam;
    switch (lpdi->itemID)
    {
        case IDM_FONT_COURIER:
            lstrcpy (buff, "Courier");
            hfont = hfontCour; break;

        case IDM_FONT_TMSRMN:
            lstrcpy (buff, "Times Roman");
            hfont = hfontTmsRmn; break;

        case IDM_FONT_HELV:
            lstrcpy (buff, "Helvetica");
            hfont = hfontHelv; break;
    }

    hfontOld = SelectObject (lpdi->hDC, hfont);

/*
Если текущий пункт меню должен быть подсвечен,
установить правильные системные цвета
*/
    if (lpdi->itemState & ODS_SELECTED)
    {
        crFore = SetTextColor (lpdi->hDC, crHighlightText);
        crBack = SetBkColor (lpdi->hDC, crHighlight);
        fSwapColors = TRUE;
    }

    else fSwapColors = FALSE;

/*
Взять ширину контрольных маркеров
*/
    dwCheckMark = GetMenuCheckMarkDimensions();
    cxCheck = LOWORD (dwCheckMark);
    xText = lpdi->rcItem.left + cxCheck;
    yText = lpdi->rcItem.top;

```



```
/*
Вывод строки текста
*/
```

```
ExtTextOut (lpdi->hDC,
            xText,
            yText,
            ETO_OPAQUE,
            &lpdi->rcItem,
            buff,
            strlen(buff),
            NULL);
```

```
/*
Восстановить контекст устройства перед тем, как послать его обратно
*/
```

```
// Восстановить шрифт
SelectObject (lpdi->hDC, hfontOld);
if (fSwapColors)
{
    // Восстановить цвета
    SetTextColor (lpdi->hDC, crFore);
    SetBkColor (lpdi->hDC, crBack);
}
}
```

```
/*
Компонентная функция класса TOwnDrawWindow
*/
```

```
void TOwnDrawWindow::WMMeasureItem(TMessage& Msg)
{
    char buff[80];
    DWORD dwSize;
    DWORD dwCheckMark;
    HDC hdc;
    HFONT hfont;
    int cxWidth, cyHeight;
    int cxCheck, cyCheck;
    LPMEASUREITEMSTRUCT lpmi;

    lpmi = (LPMEASUREITEMSTRUCT)Msg.LParam;
    switch (lpmi->itemID)
    {
        case IDM_FONT_COURIER:
            strcpy (buff, "Courier");
            hfont = hfontCour;
            break;

        case IDM_FONT_TMSRMN:
            strcpy (buff, "Times Roman");
            hfont = hfontTmsRmn;
            break;

        case IDM_FONT_HELV:
```



```

        lstrcpy (buff, "Helvetica");
        hfont = hfontHelv;
        break;
    }

    /* Нахождение размера строки */
    hdc = GetDC (HWindow);
    SelectObject (hdc, hfont);
    dwSize = GetTextExtent (hdc, buff, lstrlen(buff));
    ReleaseDC (HWindow, hdc);

    /* Начать с информации о размере строки */
    cxWidth = LOWORD(dwSize);
    cyHeight = HIWORD(dwSize);

    /* Добавить информацию о размере контрольных маркеров */
    dwCheckMark = GetMenuCheckMarkDimensions();
    cxCheck = LOWORD (dwCheckMark);
    cyCheck = HIWORD (dwCheckMark);

    lpmi->itemWidth = cxWidth + cxCheck;
    lpmi->itemHeight = max (cyHeight, cyCheck);
}

```

/*
OWNDRAW.CPP - Пример собственного меню
 */

```

#include <owl.h>
#include "owndraw.h"

#define max(a,b)      (((a) > (b)) ? (a) : (b))
#define COMMANDMSG(arg) (arg.WParam)

/*
Описания классов
*/

class TOwnDrawApplication : public TApplication
{
public:
    TOwnDrawApplication (LPSTR lpszName,
                        HANDLE hInstance,
                        HANDLE hPrevInstance,
                        LPSTR lpszCmdLine,
                        int nCmdShow);

    virtual void InitMainWindow ();
};

class TOwnDrawWindow : public TWindow
{
public:
    TOwnDrawWindow (PTWindowsObject pwParent,
                    LPSTR lpszTitle,
                    PTModule pmModule);

    ~TOwnDrawWindow();
}

```



```

virtual LPSTR GetClassName ();
virtual void GetWindowClass (WNDCLASS&);
virtual void WMCommand(TMessage& Msg)    = [WM_COMMAND];
virtual void WMCreate(TMessage& Msg)     = [WM_CREATE];
virtual void WMDrawItem(TMessage& Msg)    = [WM_DRAWITEM];
virtual void WMMeasureItem (TMessage& Msg) = [WM_MEASUREITEM];

private:
    HFONT hfontTmsRmn;
    HFONT hfontHelv;
    HFONT hfontCour;
    COLORREF crHighlight;
    COLORREF crHighlightText;
};

/*
Функция WinMain
*/

int PASCAL WinMain(HANDLE hInstance,
                  HANDLE hPrevInstance,
                  LPSTR lpszCmdLine,
                  int nCmdShow)

{
    TOwnDrawApplication OwnDraw ("OwnDraw",
                                  hInstance,
                                  hPrevInstance,
                                  lpszCmdLine,
                                  nCmdShow);

    OwnDraw.Run();
    return OwnDraw.Status;
}

/*
Компонентная функция класса прикладной программы
*/
TOwnDrawApplication::TOwnDrawApplication
    (LPSTR lpszName,
     HANDLE hInstance,
     HANDLE hPrevInstance,
     LPSTR lpszCmdLine,
     int nCmdShow )
    :TApplication(lpszName,
                  hInstance,
                  hPrevInstance,
                  lpszCmdLine,
                  nCmdShow)

{
    /* Здесь инициализируется прикладная программа. */
}

/*
Компонентная функция класса прикладной программы:
*/

void TOwnDrawApplication::InitMainWindow ()

```



```

{
    MainWindow = new TOwnDrawWindow (NULL, "Owner Draw Menu", NULL);
}

/*
Компонентная функция класса TOwnDrawWindow
*/
TOwnDrawWindow::TOwnDrawWindow (PTWindowsObject pwParent,
                                  LPSTR lpszTitle,
                                  PTModule pmModule)

    :TWindow (pwParent,
              lpszTitle,
              pmModule)
{
    HDC          hdc;
    LOGFONT      lf;
    TEXTMETRIC   tm;

    /*
    Создание трех логических шрифтов с различными размерами и стилями
    sizes and styles
    */

    memset (&lf, 0, sizeof (LOGFONT));

    hdc = CreateDC ("DISPLAY", 0, 0, 0);
    GetTextMetrics (hdc, &tm);
    DeleteDC (hdc);

    // Шрифт 1: Bold Courier.
    lf.lfWeight = 700;
    lstrcpy ((LPSTR)&lf.lfFaceName[0], "Courier");
    lf.lfHeight = tm.tmHeight;
    hfontCour = CreateFontIndirect (&lf);

    // Шрифт 2: Times Roman.
    lstrcpy ((LPSTR)&lf.lfFaceName[0], "Tms Rmn");
    lf.lfHeight = tm.tmHeight + tm.tmHeight/4;
    hfontTmsRmn = CreateFontIndirect (&lf);

    // Шрифт 3: Helv.
    lstrcpy ((LPSTR)&lf.lfFaceName[0], "Helv");
    lf.lfHeight = tm.tmHeight + tm.tmHeight/2;
    hfontHelv = CreateFontIndirect (&lf);

    /*
    Выбор системных цветов для прорисовки пунктов меню
    */

    crHighlight = GetSysColor (COLOR_HIGHLIGHT);
    crHighlightText = GetSysColor (COLOR_HIGHLIGHTTEXT);
}

/*
Компонентная функция класса TOwnDrawWindow
*/
TOwnDrawWindow::~TOwnDrawWindow ()

```



```

    {
        DeleteObject (hfontTmsRmn);
        DeleteObject (hfontHelv);
        DeleteObject (hfontCour);
    }

/*
Компонентная функция класса TOwnDrawWindow
*/

LPSTR TOwnDrawWindow::GetClassName ()
{
    return "OwnDraw:MAIN";
}

/*
Компонентная функция класса TOwnDrawWindow
*/

void TOwnDrawWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
    wc.lpszMenuName = "#1";
}

/*
Компонентная функция класса TOwnDrawWindow
*/

void TOwnDrawWindow::WMCommand(TMessage& Msg)
{
    char buffer[80];

    if (COMMANDMSG(Msg) == IDM_FILE_EXIT)
        SendMessage (HWindow, WM_SYSCOMMAND, SC_CLOSE, 0L);
    else
    {
        wsprintf(buffer, "Command = %d", COMMANDMSG(Msg));
        MessageBox(HWindow, buffer, "WM_COMMAND", MB_OK);
    }
}

/*
Компонентная функция класса TOwnDrawWindow
*/

void TOwnDrawWindow::WMCreate(TMessage& Msg)
{
    HMENU    hmenu;
    HMENU    hmenuPopup;

    /*
    Создание трех собственных пунктов меню
    */

    hmenu = GetMenu (Msg.Receiver);
    hmenuPopup = CreatePopupMenu();
    AppendMenu (hmenu, MF_POPUP, hmenuPopup, "Fo&nt");

```



```
AppendMenu (hmenuPopup,
            MF_OWNERDRAW,
            IDM_FONT_COURIER,
            MAKEINTRESOURCE(IDM_FONT_COURIER));
```

```
AppendMenu (hmenuPopup,
            MF_OWNERDRAW,
            IDM_FONT_TMSRMN,
            MAKEINTRESOURCE(IDM_FONT_TMSRMN));
```

```
AppendMenu (hmenuPopup,
            MF_OWNERDRAW,
            IDM_FONT_HELV,
            MAKEINTRESOURCE(IDM_FONT_HELV));
```

```
}
```

```
/*
```

Компонентная функция класса TOwnDrawWindow

```
*/
```

```
void TOwnDrawWindow::WMDrawItem(TMessage& Msg)
```

```
{
    char buff[80];
    COLORREF crFore;
    COLORREF crBack;
    DWORD dwCheckMark;
    HFONT hfontOld;
    HFONT hfont;
    int fSwapColors;
    int xText,yText;
    int cxCheck;
    LPDRAWITEMSTRUCT lpdi;
```

```
    lpdi = (LPDRAWITEMSTRUCT)Msg.LParam;
```

```
    switch (lpdi->itemID)
```

```
    {
```

```
        case IDM_FONT_COURIER:
```

```
            lstrcpy (buff, "Courier");
```

```
            hfont = hfontCour;
```

```
            break;
```

```
        case IDM_FONT_TMSRMN:
```

```
            lstrcpy (buff, "Times Roman");
```

```
            hfont = hfontTmsRmn;
```

```
            break;
```

```
        case IDM_FONT_HELV:
```

```
            lstrcpy (buff, "Helvetica");
```

```
            hfont = hfontHelv;
```

```
            break;
```

```
    }
```

```
    hfontOld = SelectObject (lpdi->hDC, hfont);
```

```
    /*
```

**Если текущий пункт меню должен быть подсвечен,
установить правильные системные цвета**

```
    */
```

```
    if (lpdi->itemState & ODS_SELECTED)
```

```
    {
```



```

        crFore = SetTextColor (lpdi->hDC, crHighlightText);
        crBack = SetBkColor (lpdi->hDC, crHighlight); fSwapColors = TRUE;
    }

    else
        fSwapColors = FALSE;

/*
Взять ширину контрольных маркеров
*/

    dwCheckMark = GetMenuCheckMarkDimensions();
    cxCheck = LOWORD (dwCheckMark);
    xText = lpdi->rcItem.left + cxCheck;
    yText = lpdi->rcItem.top;

/*
Вывод строки текста
*/
    ExtTextOut (lpdi->hDC,
                xText,
                yText,
                ETO_OPAQUE,
                &lpdi->rcItem,
                buff,
                lstrlen(buff),
                NULL);

/*
Восстановить контекст устройства перед тем, как послать его обратно
*/

// Восстановить шрифт
SelectObject (lpdi->hDC, hfontOld);
if (fSwapColors)
{
    // Восстановить цвета
    SetTextColor (lpdi->hDC, crFore);
    SetBkColor (lpdi->hDC, crBack);
}
}

/*
Компонентная функция класса TOwnDrawWindow
*/

void TOwnDrawWindow::WMMeasureItem(TMessage& Msg)
{
    char buff[80];
    DWORD dwSize;
    DWORD dwCheckMark;
    HDC hdc;
    HFONT hfont;
    int cxWidth, cyHeight;
    int cxCheck, cyCheck;
    LPMEASUREITEMSTRUCT lpmi;

```



```

lpmi = (LPMEASUREITEMSTRUCT)Msg.LParam;
switch (lpmi->itemID)
{
    case IDM_FONT_COURIER:
        lstrcpy (buff, "Courier");
        hfont = hfontCour;
        break;

    case IDM_FONT_TMSRMN:
        lstrcpy (buff, "Times Roman");
        hfont = hfontTmsRmn;
        break;

    case IDM_FONT_HELV:
        lstrcpy (buff, "Helvetica");
        hfont = hfontHelv;
        break;
}

/* Нахождение размера строки */
hdc = GetDC (HWindow);
SelectObject (hdc, hfont);
dwSize = GetTextExtent (hdc, buff, lstrlen(buff));
ReleaseDC (HWindow, hdc);

/* Начать с информации о размере строки */
cxWidth = LOWORD(dwSize);
cyHeight = HIWORD(dwSize);

/* Добавить информацию о размере контрольных маркеров. */
dwCheckMark = GetMenuCheckMarkDimensions();
cxCheck = LOWORD (dwCheckMark);
cyCheck = HIWORD (dwCheckMark);

lpmi->itemWidth = cxWidth + cxCheck;
lpmi->itemHeight = max (cyHeight, cyCheck);
}

```

BITMENU.H

```

/*
Bitmenu.H - подключаемый файл для BITMAP.CPP
*/

#define IDM_FILE_NEW      100
#define IDM_FILE_OPEN     101
#define IDM_FILE_SAVE     102
#define IDM_FILE_SAVEAS  103
#define IDM_FILE_PRINT    104
#define IDM_FILE_EXIT     105

#define IDM_EDIT_UNDO     200
#define IDM_EDIT_CUT      201
#define IDM_EDIT_COPY     202
#define IDM_EDIT_PASTE    203
#define IDM_EDIT_CLEAR    204
#define IDM_EDIT_DELETE   205

```



```
#define IDM_MENU_FREE 300
```

```
BITMENU.RC
```

```
#include "bitmenu.h"
```

```
snapshot icon BitMenu.ico
```

```
hand cursor BitMenu.cur
```

```
1 MENU
```

```
{  
  POPUP "&File"  
  {  
    MENUITEM "&New",           IDM_FILE_NEW  
    MENUITEM "&Open...",       IDM_FILE_OPEN  
    MENUITEM "&Save",          IDM_FILE_SAVE  
    MENUITEM "Save &As...",    IDM_FILE_SAVEAS  
    MENUITEM SEPARATOR  
    MENUITEM "&Print",         IDM_FILE_PRINT  
    MENUITEM SEPARATOR  
    MENUITEM "E&xit",         IDM_FILE_EXIT  
  }  
  
  POPUP "&Edit"  
  {  
    MENUITEM "&Undo\tAlt+Backspace", IDM_EDIT_UNDO  
    MENUITEM SEPARATOR  
    MENUITEM "Cu&t\tShift+Del",      IDM_EDIT_CUT  
    MENUITEM "&Copy\tCtrl+Ins",      IDM_EDIT_COPY  
    MENUITEM "&Paste\tShift+Ins",    IDM_EDIT_PASTE  
    MENUITEM SEPARATOR  
    MENUITEM "Cl&ear",              IDM_EDIT_CLEAR  
    MENUITEM "&Delete",              IDM_EDIT_DELETE  
  }  
}
```

```
BITMENU.DEF
```

```
NAME BITMENU
```

```
EXETYPE WINDOWS
```

```
DESCRIPTION 'BitMenu - Bitmap in a menu'
```

```
CODE MOVEABLE DISCARDABLE
```

```
DATA MOVEABLE MULTIPLE
```

```
HEAPSIZE 512
```

```
STACKSIZE 5120
```

Большинство программ Bitmenu взаимодействует с двумя функциями, входящими в объект окна: конструктором и функцией отклика на сообщение для сообщения WM_CREATE, WMCreate(). Так же, как и в случае элемента меню пользовательской прорисовки, возможно, было бы лучше создать отдельный "объект-меню битового образа", чтобы избавить объект окна от сложностей меню. Для того чтобы Вы лучше разобрались в применении битовых образов в меню, они приведены вместе.

В своем конструкторе объект окна Bitmenu начинает с обращения к GetSystemMetrics для определения установок размера пиктограммы драйвера экрана. Эти величины используются для вычисления размера битовых образов. Как мы отмечали ранее, разумнее всего избегать жесткого кодирования размеров, поскольку та-

кой подход может привести к непредсказуемым результатам на дисплеях высокого разрешения. Bitmenu создает битовые образы, размер которых относителен, оставаясь пропорциональным системным пиктограммам:

```
cxBitmapWigth  = GetSystemMetrics(SM_CXICON) * 3;  
cxBitmapHeight = GetSystemMetrics(SM_CYICON);
```

Когда создан битовый образ памяти, GDI не инициализирует для поверхности прорисовки какой-либо особый цвет. До того, как мы сможем отрисовать битовый образ, мы должны очистить область прорисовки во избежание появлений случайных цветов. Программа Owndraw вызывает GetSysColors для выяснения, какой цвет следует использовать для текущего меню, Bitmenu осуществляет такой же вызов и использует полученный в результате цвет для создания закрашки, с помощью которой очищает фон для битового образа:

```
crBackground = GetSysColor(COLOR_MENU);  
hbrBackground = CreateSolidBrush(crBackground);
```

По опыту обсуждения программы Owndraw Вы знаете, что мы можем обратиться к GDI для прорисовки битового образа лишь тогда, когда создан контекст устройства и когда он связан с битовым образом. Мы вызываем CreateDC для получения экранного контекста устройства, затем модифицируем данный контекст устройства для использования с битовым образом путем обращения к CreateCompatibleDC. Далее мы устанавливаем фоновую закрашку в контекст устройства, после чего он готов к тому, чтобы инициализировать битовый образ:

```
hdcScreen = CreateDC("DISPLAY", 0, 0, 0);  
hdcBitmap = CreateCompatibleDC(hdcScreen);  
hbrOld = SelectObject(hdcBitmap, hbrBackground);
```

Затем конструктор объекта окна создает битовый образ и инициализирует его в цвет той закрашки, которую мы только что создали - hbrBackground. После того, как каждый битовый образ будет создан, PatBlt заполнит каждый битовый образ цветами закрашки:

```
for(i=0;i<COUNT;j++)  
{  
    hbm[i] = CreateCompatibleBitmap(hdcScreen,  
                                    cxWigth,  
                                    cyHeight);  
  
    if(i == 0)  
        ld = SelectObject(hdcBitmap, hbm[0]);  
    else  
        SelectObject(hdcBitmap, hbm[i]);  
    PatBlt(hdcBitmap,0,0, cxWigth, cyHeight, PATCOPY);  
}
```

Этот код дает важную информацию о GDI. При создании контекстов устройств Вы должны восстанавливать первоначальное состояние контекста устройства перед тем, как его удалить. По этой причине данный код запоминает в hbmOld идентификатор первого битового образа, выбранного из контекста устройства. Перед тем как ликвидировать контекст устройства, мы будем выбирать этот битовый образ снова в контекст устройства. Когда битовые образы инициализированы, мы очистим все созданные объекты GDI:

```
SelectObject(hdcBitmap, hbmOld);  
SelectObject(hdcBitmap, hbrOld);  
DeleteDC(hdcBitmap);  
DeleteDC(hdcScreen);
```



```
DeleteObject(hbrBackground);
```

Конструктор полностью отвечает за создание закрасок для различных последовательностей битового образа:

```
hbr[0] = CreateHatchBrush(HS_DIAGCROSS, RGB(0, 0, 0));  
hbr[1] = CreateSolidBrush(RGB(64, 64, 64));  
hbr[2] = CreateHatchBrush(HS_CROSS, RGB(0, 0, 0));  
hbr[3] = CreateSolidBrush(RGB(0, 0, 0));  
hbr[4] = CreateHatchBrush(HS_VERTICAL, RGB(0, 0, 0));
```

После этого конструктор инициализирует каждый элемент данных в объекте окна Bitmenu. Эти данные готовы и ждут использования подобно этапной бригаде, ожидающей начала автогонок. Сообщение WM_CREATE служит стартовым флажком для запуска объекта окна, а этапная бригада бросается устанавливать всплывающее меню с несколькими битовыми образами. Прежде всего создается контекст устройства, предназначенный для использования с битовыми образами:

```
hdcScreen = GetDC(HWindow);  
hdcBitmap = CreateCompatibleDC(hdcScreen);  
ReleaseDC(HWindow, hdcScreen);
```

Затем создается пустое всплывающее меню и присоединяется к меню верхнего уровня:

```
hmenuPopup = CreatePopupMenu();  
hmenu = GetMenu(hwnd);  
AppendMenu(hmenu, MF_POPUP, hmenuPopup, "&Brushes");
```

Так же, как и до этого, мы сохраняем копию исходных объектов GDI в нашем контексте устройства для использования при удалении контекста устройства:

```
hbrOld = SelectObject(hdcBitmap, hbr[0]);  
hbmOld = SelectObject(hdcBitmap, hbm[0]);
```

После этого, вызывая SelectObject, мы перебираем все битовые образы, чтобы установить связь каждого битового образа с контекстом устройства битового образа. Второй раз SelectObject вызывается для того, чтобы связать контекст устройства с типом закрашки. Сделав это, мы прорисовываем образ закрашки в битовом образе с помощью процедуры Rectangle. Как Вы помните, эта процедура использует для прорисовки границ установленное в данный момент перо, а для внутренней части прямоугольника использует текущую установленную закрашку. После того как это сделано, битовый образ готов к тому, чтобы его включили в меню с помощью процедуры AppendMenu:

```
for (i=0; i<COUNT; i++)  
{  
    SelectObject(hdcBitmap, hbr[i]);  
    SelectObject(hdcBitmap, hbm[i]);  
    Rectangle(hdcBitmap, 0, cyOffset, cxWidth, cyHeight-cyOffset);  
    AppendMenu(hmenuPopup,  
                MF_BITMAP,  
                IDM_FIRST+1,  
                MAKEINTRESOURCE(hbm[i]));  
}
```

Как только битовый образ создан и включен в меню, мы должны очистить созданные объекты GDI. Единственный объект, о котором мы должны беспокоиться, это контекст устройства битового образа. Прежде всего нам необходимо расчлени-

связь между закрасками и битовыми образами, которые вновь созданы. Лишь после этого контекст устройства можно благополучно ликвидировать, без опасения оставить объекты в системе на произвол судьбы:

```
SelectObject(hdcBitmap, GetStockObject(BLACK_BRUSH));  
SelectObject(hdcBitmap, hbmOld);  
DeleteObject(hbrBackground);
```

После этого в памяти остается два комплекта объектов GDI - битовые образы и их закраски. Объект нашего окна сохраняет эти объекты на протяжении всего своего существования. Битовые образы нельзя разрушить до тех пор, пока не разрушено меню, в котором они используются. С другой стороны, объекты закраски сохраняются, поскольку они могут понадобиться при закраске объектов, выбранных пользователем из элементов меню битового образа.

Эти объекты удаляются в конце концов деструктором объекта окна:

```
for (i=0; i<COUNT; i++)  
{  
    DeleteObject(hbr[i]);  
    DeleteObject(hbm[i]);  
}
```

Использование битовых образов в меню и элементов меню пользовательской прорисовки - это два способа, с помощью которых Вы можете присовокуплять графические объекты к своему меню. Теперь мы переходим к рассмотрению третьего способа: контрольные маркеры владельца.

Создание контрольных маркеров меню владельца

В последней главе мы упоминали о том, что контрольные маркеры обеспечивают один тип визуального ключа в меню. Несмотря на то, что стандартный контрольный маркер полезен во многих ситуациях, Вам, возможно, захочется использовать в своем меню различные символы. Вы можете обратиться к процедурам GDI, чтобы создать свои собственные контрольные маркеры владельца и установить их в меню. Когда Вы станете это делать, система меню предоставит возможность добавить по два различных битовых образа к каждому элементу меню. Один битовый образ будет выводиться на экран тогда, когда элемент меню находится в контролируемом состоянии, другой - предназначен для вывода при неконтролируемом состоянии.

Создание битового образа контрольного маркера начинается с создания обычного битового образа GDI. Как это делается, мы обсуждали раньше в контексте программы Bitmenu в этой главе. Вы должны создать битовый образ, создать контекст устройства, а затем связать их вместе. После этого Вы используете процедуры GDI для прорисовки каких угодно фигур, используемых для отличия контролируемого и неконтролируемого состояний. Требуется большая аккуратность, поскольку прорисовка должна быть достаточно малой для того, чтобы поместиться в пространстве, отведенном системой меню под контрольные маркеры.

Для того чтобы определить разрешенный для работы размер, Вы обращаетесь к процедуре GetMenuCheckMarkDimensions. Эта процедура не имеет параметров, зато передает значение DWORD (длинное без знака), в котором упакованы необходимые размеры битового образа контрольного маркера. Ниже показано, как можно извлечь из этой величины информацию о размерах:

```
DWORD dwCheck;  
int cxWigth, cyHeight;  
  
dwCheck = GetMenuCheckMarkDimensions();  
cxWigth = LOWORD(dwCheck);  
cyHeight = HIWORD(dwCheck);
```


Используя информацию о размерах с помощью этой процедуры, Вы можете создать пару битовых образов, задать подходящую прорисовку контролируемого и не контролируемого состояний и присоединить битовый образ к особым элементам меню, которые по Вашему желанию будут использовать контрольные маркеры. После того как битовые образы станут равноправными объектами GDI, Вы сможете создать отдельную пару битовых образов контрольного маркера и использовать их во многих других элементах меню. Для присоединения битового образа к элементу меню необходимо воспользоваться процедурой `SetMenuItemBitmap`, которая имеет следующий вид:

```
BOOL SetMenuItemBitmap(hMenu, nPosition, wFlags,
                      hbmUnchecked, hbmChecked)
```

где

- * `hMenu` - идентификатор меню;
- * `nPosition` определяет элемент меню, к которому присоединяется пара битовых образов. Если следующий параметр `wFlags` - `MF_BYCOMMAND`, то `nPosition` - результирующий код команды. Если значение следующего параметра `MF_BYPOSITION`, то `nPosition` представляет относительную позицию элемента меню в меню с идентификатором `hMenu`;
- * `wFlags` принимает значение либо `MF_BYCOMMAND`, либо `MF_BYPOSITION`;
- * `hbmUnchecked` - идентификатор битового образа GDI, который выводится на экран при неконтролируемом состоянии элемента меню;
- * `hbmChecked` - идентификатор битового образа GDI, который выводится на экран при контролируемом состоянии элемента меню.

Как только битовые образы контрольного маркера присоединяются к элементу меню, они используются автоматически, без дополнительных усилий со стороны Вашей программы. Все, что требуется от Вас в этот момент, это вызвать `CheckMenuItem` для установки контролируемого состояния либо в `MF_CHECKED`, либо в `MF_UNCHECKED`.

Когда позже, в любой момент времени, Вы захотите удалить контрольный маркер владельца, следует обратиться к процедуре `SetMenuItemBitmaps`, установив значение `NULL` в параметр идентификатора битового образа, соответствующего тому битовому отображению, которое Вы хотите удалить. На рис.12.3 показано два комплекта контрольных маркеров пользователя, созданных с помощью показательной программы `Checkmenu`.

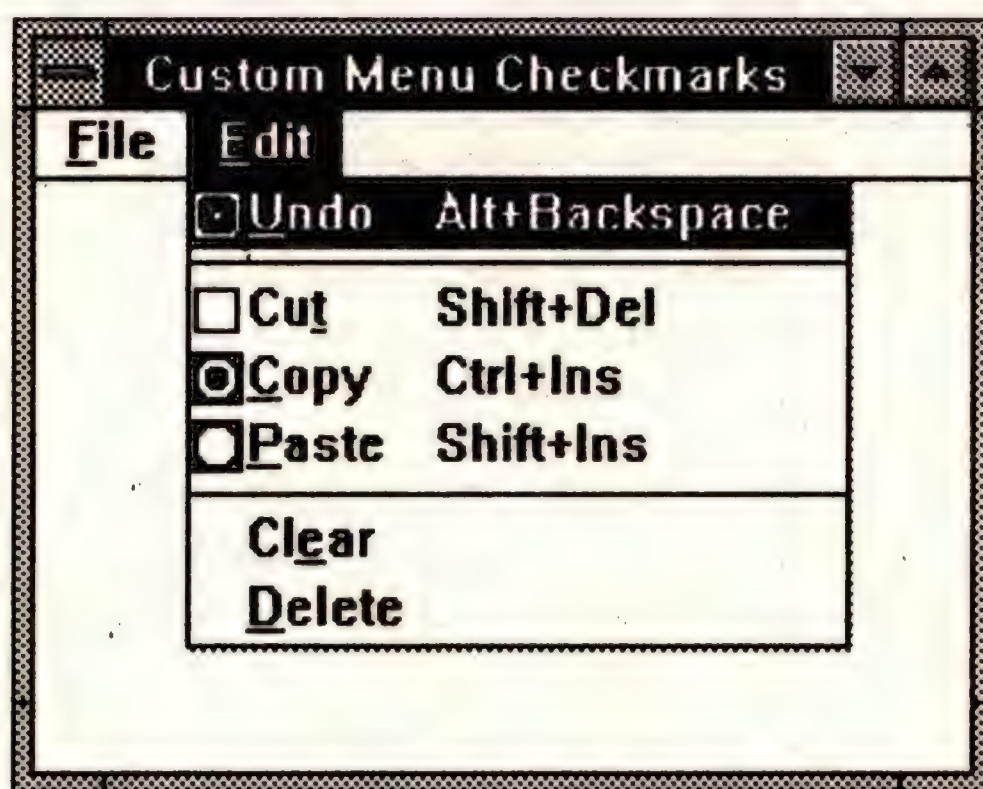


Рис.12.3. Две пары контрольных маркеров меню владельца.

Программа `Checkmenu` создает две пары контрольных маркеров меню владельца. Одна пара использует квадратную рамку с диагональным перекрестьем для контролируемого состояния и пустой квадрат для неконтролируемого состояния. Вторая пара представляет круглые контрольные маркеры с закрашенной окружностью внутри

для контролируемого состояния и пустую окружность для неконтролируемого состояния. Когда мы создадим эти две пары битовых образов контрольных маркеров, можно использовать их для любых или даже для всех элементов в нашем меню. Как Вы заметили на рис.12.3, Checkmenu использует стандартное меню, которое мы создали в последней главе.

MAKEFILE.MAK

.AUTODEPEND

Определения транслятора

```
INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;
C:\BORLANDC\INCLUDE
CC =bcc -c -D _CLASSDLL -H -ml -WS -w -IS(INC)
```

Неявные правила

c.obj:

```
$(CC) {$<}
```

cpp.obj:

```
$(CC) {$<}
```

Явные правила

```
ChekMenu.exe: ChekMenu.res ChekMenu.def ChekMenu.obj
```

```
tlink /c /C /n /P- /Twe /x @ChekMenu.LNK
```

```
RC ChekMenu.res ChekMenu.exe
```

Зависимости отдельных файлов

```
ChekMenu.obj: ChekMenu.cpp
```

```
ChekMenu.res: ChekMenu.rc ChekMenu.cur ChekMenu.ico
```

```
RC -R -FO ChekMenu.res ChekMenu.RC
```

CHEKMENU.LNK

```
c:\borlandc\lib\cOwl.obj+
```

```
ChekMenu.obj
```

```
ChekMenu,ChekMenu
```

```
\borlandc\owl\lib\owl.lib+
```

```
crtll.lib+
```

```
cwl.lib+
```

```
import.lib+
```

```
mathl.lib+
```

```
cl.lib
```

```
ChekMenu.def
```

CHEKMENU.CPP

```
/*
```

CHEKMENU.CPP - Программа, демонстрирующая пользовательские контрольные маркеры меню

```
*/
```

```
#include <owl.h>
```

```
#include "chekmenu.h"
```

```
#define COMMANDMSG(arg) (arg.WParam)
```



```

/*
Константы
*/
const int COUNT = 4;

/*
Описания классов
*/
class TChekMenuApplication : public TApplication
{
public:
    TChekMenuApplication (LPSTR lpszName,
                          HANDLE hInstance,
                          HANDLE hPrevInstance,
                          LPSTR lpszCmdLine,
                          int nCmdShow);

    virtual void InitMainWindow ();
};

class TChekMenuWindow : public TWindow
{
public:
    TChekMenuWindow (PTWindowsObject pwParent,
                    LPSTR lpszTitle,
                    PTModule pmModule);

    ~TChekMenuWindow();
    virtual LPSTR GetClassName ();
    virtual void GetWindowClass (WNDCLASS&);
    virtual void WMCommand(TMessage& Msg) = [WM_COMMAND];
    virtual void WMCreate(TMessage& Msg) = [WM_CREATE];

private:
    HANDLE hbmCheck[COUNT];
};

/*
Функция WinMain
*/
int PASCAL WinMain (HANDLE hInstance,
                   HANDLE hPrevInstance,
                   LPSTR lpszCmdLine,
                   int nCmdShow)

{
    TChekMenuApplication ChekMenu ("ChekMenu",
                                    hInstance,
                                    hPrevInstance,
                                    lpszCmdLine,
                                    nCmdShow);

    ChekMenu.Run();
    return ChekMenu.Status;
}

```



```
/*
Компонентная функция класса прикладной программы
*/
```

```
TChekMenuApplication:: TChekMenuApplication
```

```
(LPSTR IpszName,
HANDLE hInstance,
HANDLE hPrevInstance,
LPSTR IpszCmdLine,
int nCmdShow)
```

```
: TApplication(IpszName,
                hInstance,
                hPrevInstance,
                IpszCmdLine,
                nCmdShow)
```

```
{
/* Здесь инициализируется прикладная программа */
}
```

```
/*
Компонентная функция класса прикладной программы
*/
```

```
void TChekMenuApplication:: InitMainWindow ()
```

```
{
    MainWindow = new TChekMenuWindow (NULL,
                                      "Custom Menu Checkmarks",
                                      NULL);
}
```

```
/*
Компонентная функция класса TChekMenuWindow
*/
```

```
TChekMenuWindow::TChekMenuWindow
```

```
(PTWindowsObject pwParent,
LPSTR IpszTitle,
PTModule pmModule)
```

```
:TWindow (pwParent, IpszTitle, pmModule)
```

```
{
    COLORREF crBackground;
    DWORD    dwCheck;
    HBRUSH    hbrBackground;
    HBRUSH    hbrOld;
    HBITMAP   hbmOld;
    HDC       hdcScreen;
    HDC       hdcBitmap;
    int       xMin, xMax;
    int       yMin, yMax;
    int       cxCheckWidth;
    int       cyCheckHeight;
    int       cxFourth;
    int       cyFourth;
    int       cxMargin;
    int       i;
```

```
// Взять размеры контрольных маркеров
```



```

dwCheck      = GetMenuCheckMarkDimensions();
cyCheckHeight = HIWORD (dwCheck);
cxCheckWidth  = LOWORD (dwCheck);

// Подсчитывает отступ на основе ширины рамки
cxMargin = GetSystemMetrics (SM_CXBORDER);

xMin      = cxMargin;
xMax      = cxCheckWidth - cxMargin;
yMin      = cxMargin;
yMax      = cyCheckHeight - cxMargin;
cxFourth  = cxCheckWidth/4;
cyFourth  = cyCheckHeight/4;

// Создание кисти для инициализации битовых образов
crBackground = GetSysColor (COLOR_MENU);
hbrBackground = CreateSolidBrush (crBackground);

// Создание контекста устройства для установки битовых образов
hdcScreen = CreateDC ("DISPLAY", 0, 0, 0);
hdcBitmap = CreateCompatibleDC (hdcScreen);
hbrOld = SelectObject (hdcBitmap, hbrBackground);

for (i=0;i<COUNT;i++)
{
    hbmCheck[i] = CreateCompatibleBitmap (hdcScreen,
                                           cxCheckWidth,
                                           cyCheckHeight);

    if ( i == 0) hbmOld=SelectObject (hdcBitmap,hbmCheck[i]);
    else SelectObject (hdcBitmap, hbmCheck[1]);
    PatBlt (hdcBitmap, 0, 0, cxCheckWidth,
            cyCheckHeight,
            PATCOPY);
}

// Рисование первого неотмеченного пункта меню
SelectObject (hdcBitmap, hbmCheck[0]);
Rectangle (hdcBitmap, xMin, yMin, xMax, yMax);

// Рисование первого пункта отмеченного меню
SelectObject (hdcBitmap, hbmCheck[1]);
Rectangle (hdcBitmap, xMin, yMin, xMax, yMax);

MoveTo (hdcBitmap, xMin, yMin);
LineTo (hdcBitmap, xMax, yMax);
MoveTo (hdcBitmap, xMin, yMax-1);
LineTo (hdcBitmap, xMax-1, yMin);

// Рисование следующего неотмеченного пункта меню
SelectObject (hdcBitmap, hbmCheck[2]);
Ellipse (hdcBitmap, xMin, yMin, xMax, yMax);

// Рисование следующего отмеченного пункта меню
SelectObject (hdcBitmap, hbmCheck[3]);
Ellipse (hdcBitmap, xMin, yMin, xMax, yMax);
SelectObject (hdcBitmap, GetStockObject (BLACK_BRUSH));

```



```

        Ellipse (hdcBitmap,
                xMin+cxFourth,
                yMin+cyFourth,
                xMax-cxFourth,
                yMax-cyFourth);

    // Очистить созданные объекты GDI
    SelectObject (hdcBitmap, hbrOld);
    SelectObject (hdcBitmap, hbmOld);
    DeleteDC (hdcBitmap);
    DeleteObject (hbrBackground);
    DeleteDC (hdcScreen);
}

/*
Компонентная функция класса TChekMenuWindow
*/

TChekMenuWindow::~TChekMenuWindow ()
{
    int i;
    for (i=0;i<COUNT;i++) DeleteObject (hbmCheck[i]);
}

/*
Компонентная функция класса TChekMenuWindow
*/

LPSTR TChekMenuWindow::GetClassName ()
{
    return "ChekMenu:MAIN";
}

/*
Компонентная функция класса TChekMenuWindow
*/

void TChekMenuWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
    wc.lpszMenuName = "#1";
}

/*
Компонентная функция класса TChekMenuWindow
*/

void TChekMenuWindow::WMCommand(TMessage& Msg)
{
    char buffer[80];
    if (COMMANDMSG(Msg) == IDM_FILE_EXIT)
        SendMessage (HWindow, WM_SYSCOMMAND, SC_CLOSE, 0L);
    else
    {
        wsprintf(buffer, "Command = %d", COMMANDMSG(Msg));
    }
}

```



```

        MessageBox(HWindow, buffer, "WM_COMMAND", MB_OK);
    }
}

/*
Компонентная функция класса TChekMenuWindow
*/

void TChekMenuWindow::WMCreate(TMessage& Msg)
{
    HMENU hmenu;

    // Взять ключ меню верхнего уровня.
    hmenu = GetMenu (Msg.Receiver);

    // Присоединить контрольные маркеры к четырем пунктам меню
    SetMenuItemBitmaps (hmenu, IDM_EDIT_UNDO,
                        MF_BYCOMMAND, hbmCheck[0], hbmCheck[1]);
    CheckMenuItem (hmenu, IDM_EDIT_UNDO,
                  MF_BYCOMMAND | MF_CHECKED);
    SetMenuItemBitmaps (hmenu, IDM_EDIT_CUT,
                        MF_BYCOMMAND, hbmCheck[0], hbmCheck[1]);
    SetMenuItemBitmaps (hmenu, IDM_EDIT_COPY,
                        MF_BYCOMMAND, hbmCheck[2], hbmCheck[3]);
    CheckMenuItem (hmenu, IDM_EDIT_COPY,
                  MF_BYCOMMAND | MF_CHECKED);
    SetMenuItemBitmaps (hmenu, IDM_EDIT_PASTE,
                        MF_BYCOMMAND, hbmCheck[2], hbmCheck[3]);
}

```

CHEKMENU.H

```

/*
ChekMenu.H - подключаемый файл для chekmenu.cpp
*/

```

```

#define IDM_FILE_NEW      100
#define IDM_FILE_OPEN     101
#define IDM_FILE_SAVE     102
#define IDM_FILE_SAVEAS   103
#define IDM_FILE_PRINT    104
#define IDM_FILE_EXIT     105

#define IDM_EDIT_UNDO     200
#define IDM_EDIT_CUT      201
#define IDM_EDIT_COPY     202
#define IDM_EDIT_PASTE    203
#define IDM_EDIT_CLEAR    204
#define IDM_EDIT_DELETE   205

```

CHEKMENU.RC

```

#include "chekmenu.h"

snapshot icon ChekMenu.ico

hand cursor ChekMenu.cur

1 MENU
{

```


POPUP "&File"

```
{
  MENUITEM "&New",           IDM_FILE_NEW
  MENUITEM "&Open...",      IDM_FILE_OPEN
  MENUITEM "&Save",          IDM_FILE_SAVE
  MENUITEM "Save &As...",    IDM_FILE_SAVEAS
  MENUITEM SEPARATOR
  MENUITEM "&Print",         IDM_FILE_PRINT
  MENUITEM SEPARATOR
  MENUITEM "E&xit"           IDM_FILE_EXIT
}
```

POPUP "&Edit"

```
{
  MENUITEM "&Undo \tAlt+Backspace", IDM_EDIT_UNDO
  MENUITEM SEPARATOR
  MENUITEM "Cu&t \tShift+Del",      IDM_EDIT_CUT
  MENUITEM "&Copy \tCtrl+Ins",      IDM_EDIT_COPY
  MENUITEM "&Paste \tShift+Ins",    IDM_EDIT_PASTE
  MENUITEM SEPARATOR
  MENUITEM "Cl&ear",               IDM_EDIT_CLEAR
  MENUITEM "&Delete",              IDM_EDIT_DELETE
}
```

CHEKMENU.DEF

NAME CHEKMENU

EXETYPE WINDOWS

DESCRIPTION 'ChekMenu - Custom menu checkmarks'

CODE MOVEABLE DISCARDABLE

DATA MOVEABLE MULTIPLE

HEAPSIZE 512

STACKSIZE 5120

Программа Checkmenu комбинирует функциональные возможности меню в объекте окна. Так же, как и в других программах в этой главе, это сделано для концентрации внимания на исходных положениях API Windows, а не демонстрации принципов объектно-ориентированного проектирования.

Инициализация и создание контрольных маркеров в меню проходит в два этапа. В конструкторе объекта окна создаются битовые образы как для контролируемого, так и неконтролируемого состояний. В ответ на сообщение WM_CREATE, эти битовые образы прикрепляются к меню и при этом, для демонстрации, некоторые элементы меню являются контролируруемыми, а некоторые нет.

Конструктор объекта окна начинает с опроса системы, выясняя необходимый размер контрольного маркера:

// Получение информации о размерах контрольного маркера.

```
dwCheck = GetMenuCheckMarkDimensions();
cyHeight = HIWORD(dwCheck);
cxWigth = LOWORD(dwCheck);
```

Имея эту информацию, программа Checkmenu вычисляет рамку по периметру нашего контрольного маркера. Это необходимо, поскольку данная область фактически касается левой границы меню. Фиксация границ позволяет избежать возможной тес-

ноты. Мы вычисляем рамку вызовом `GetSystemMetrics` с параметром `SM_CXBORDER` для получения ширины тонкой рамки окна при использовании в качестве границ контрольного маркера:

```
/* Вычисление границ на базе ширины рамки */  
cxMargin = GetSystemMetrics (SM_CXBORDER);  
xMin     = cxMargin;  
xMax     = cxWigth - cxMargin;  
yMin     = cyMargin;  
yMax     = cyHeight - cxMargin;  
cxFourth = cxWigth/4;  
cyFourth = cyHeight/4;
```

Для создания битового образа необходимо инициализировать его фон. Для фоновго цвета меню мы вызовем `GetSysColor`, затем станем использовать этот цвет для создания закрашки:

```
/*Создание фоновой закрашки для инициализации битового отображения */  
crBackground = GetSysColor (COLOR_MENU);  
hbrBackground = CreateSolidBrush (crBackground);
```

Как видно из программы `Bitmenu`, для прорисовки битового образа необходим контекст устройства. Далее показано, как `Checkmenu` создает контекст устройства, набор битовых образов и использует фоновую закрашку для инициализации наших битовых образов в фоновый цвет меню:

```
/* Создание контекста устройства и битовых отображений */  
hdcScreen = GetDC(hwnd);  
hdcBitmap = CreateCompatibleDC(hdcScreen);  
hbrOld = SelectObject(hdcBitmap, hbrBackground);  
for (i=0;i<COUNT;j++)  
{  
    hbmCheck[i]=CreateCompatibleBitmap (hdcScreen, cxWigth, cyHeight);  
    if (i == 0)  
        hbmOld = SelectObject (hdcBitmap, hbmCheck[0]);  
    else  
        SelectObject (hdcBitmap, hbmCheck[i]);  
    PatBlt (hdcBitmap,0,0, cxWigth, cyHeight, PATCOPY);  
}  
ReleaseDC (hwnd, hdcScreen);
```

Прорисовка четырех образов контрольного маркера так же проста, как и вызов нескольких процедур GDI. Конечно, наших четыре битовых образа используют контекст устройства, поэтому необходима процедура `SelectObject` для установления связи с каждым битовым образом:

```
// Прорисовка первого неконтролируемого элемента меню  
SelectObject (hdcBitmap, hbmCheck[0]);  
Rectangle (hdcBitmap, xMin, yMin, xMax, yMax);  
  
// Прорисовка первого контролируемого элемента меню  
SelectObject (hdcBitmap, hbmCheck[1]);  
Rectangle (hdcBitmap, xMin, yMin, xMax, yMax);  
MoveTo (hdcBitmap, xMin, yMin)
```



```

LineTo (hdcBitmap, xMax, yMax)
MoveTo (hdcBitmap, xMin, yMax-1)
LineTo (hdcBitmap, xMax-1, yMin)

// Прорисовка второго неконтролируемого элемента меню
SelectObject (hdcBitmap, hbmCheck[2]);
Ellipse (hdcBitmap, xMin, yMin, xMax, yMax);

// Прорисовка второго контролируемого элемента меню
SelectObject (hdcBitmap, hbmCheck[3]);
Ellipse (hdcBitmap, xMin, yMin, xMax, yMax);
SelectObject (hdcBitmap, GetStockObject (BLACK_BRUSH));
Ellipse (hdcBitmap,
        xMin+cxFourth,
        yMin+cyFourth,
        xMax-cxFourth,
        yMax-cyFourth);

```

Работа конструктора завершается очисткой всех объектов GDI, которые были попутно созданы. Сюда же включается проверка первоначального удаления связей с какими бы то ни было объектами GDI:

```

// Очистка объектов GDI, которые мы создали.
SelectObject(hdcBitmap, hbrOld);
SelectObject(hdcBitmap, hbmOld);
DeleteDC(hdcBitmap);
DeleteObject(hbrBackground);
DeleteDC(hdcScreen);

```

Контрольные маркеры меню прикрепляются к меню объекта нашего окна в ответ на сообщение WM_CREATE. Но прежде необходим идентификатор для меню верхнего уровня нашего окна:

```

// Получение идентификатора меню верхнего уровня
hmenu = GetMenu (Msg.Receiver);

```

Затем происходит четыре обращения к SetMenuItemBitmap:

```

// Прикрепление контрольных маркеров к четырём элементам меню
SetMenuItemBitmap (hmenu, IDM_EDIT_UNDO, MF_BYCOMMAND,
                  hbmCheck[0], hbmCheck[1];
CheckMenuItem (hmenu, IDM_EDIT_UNDO,
               MF_BYCOMMAND | MF_CHECKED);
SetMenuItemBitmap (hmenu, IDM_EDIT_CUT, MF_BYCOMMAND,
                  hbmCheck[0], hbmCheck[1];
SetMenuItemBitmap (hmenu, IDM_EDIT_COPY, MF_BYCOMMAND,
                  hbmCheck[2], hbmCheck[3];
CheckMenuItem (hmenu, IDM_EDIT_COPY,
               MF_BYCOMMAND | MF_CHECKED);
SetMenuItemBitmap (hmenu, IDM_EDIT_PASTE, MF_BYCOMMAND,
                  hbmCheck[2], hbmCheck[3];

```

Теперь мы можем контролировать или не контролировать элементы меню, а система меню позаботится о выводе на экран корректных битовых образов.

Но есть ещё один момент. Поскольку мы создали битовые образы GDI, следует убедиться в том, что мы очистили их после использования. Раз уж мы создавали их в конструкторе объекта окна, весьма существенно ликвидировать их в деструкторе объекта окна. Именно так и происходит в Checkmenu:

```
TCheckMenuWindow::~TCheckMenuWindow()
{
    int i;
    for (i=0;i<COUNT;i++)
        DeleteObject (hbmCheck[i]);
}
```

Создание контрольных маркеров владельца - прямолинейное решение, хотя оно требует понимания того, как создавать и как манипулировать битовыми образами GDI. Когда битовый образ создан и связан с контекстом устройства, прорисовка битового образа сводится к обращению в различные процедуры прорисовки GDI, в том числе и к PatBlt для очистки фона. Затем Rectangle, Ellipse, MoveTo и LineTo прорисуют фактический образ контрольного маркера. Как мы показали в примерах программ, очень важно очищать объекты прорисовки GDI, которые Вы создали.

В следующей главе мы вернемся к тому, о чем мы впервые упоминали в главе 2, где давалось введение в программу-минимум для Windows. Итогом стало создание окна. Программа не в состоянии создать более одного удобного окна, и тогда наступают моменты, когда для хорошего пользовательского интерфейса действительно необходимо гораздо больше окон. И даже если Вы в своих оконных программах не создадите более одного окна, знайте, что некоторые специальные возможности Windows скрыты в самом процессе создания окна. Давайте же разберемся, в чем же заключаются эти возможности.

Приложение С. Словарь терминов

_AHINCR

Это глобальный символ, который используется прикладной программой для увеличения значения селектора сегмента при адресации объектов памяти, занимающих более одного сегмента.

32-Bit Addressing — 32-битовая адресация

Данный термин относится к процессору Intel 80386 и более поздним моделям, которые используют 32-х битовое смещение при адресации памяти. Это позволяет создавать сегменты памяти размером до 4 Гбайт. Когда Windows работает в режиме 386-Enhanced, этот тип адресации используется внутренне, но он не доступен непосредственно прикладной программе. Однако фирма Microsoft обеспечивает 32-битовую адресацию в библиотеке динамической компоновки, которая включена в набор инструментальных средств разработки программного обеспечения Windows с именем WINMEM32.

ANSI

Американский национальный институт стандартов.

ANSI Character Set — Набор символов ANSI

Стандарт, который определяет связь между изображениями символов и их цифровыми кодами. В соответствии со стандартом ANSI значения от 0 до 31 (1Fh) зарезервированы для управляющих кодов, значения от 32 (20h) до 127 (7Fh) - для стандартного набора символов печати, а значения от 128 (80h) до 255 (FFh) задают диапазон, в котором можно определять собственные последовательности символов. Семейство компьютеров фирмы IBM определяет таблицу кодов, которая содержит набор символов рисования линий, букв греческого алфавита и других символов. Windows определяет более сложный набор дополнительных символов, позволяющих более просто использовать программы Windows в "неанглоязычных" странах.

ASCII

Американский стандартный код для обмена информацией.

Application Message Queue — Очередь сообщений приложения

При работе приложения Windows выделяет буфер, который используется для хранения направленных (с помощью программы PostMessage) сообщений. Эти сообщения ожидают в очереди, пока программа не вызовет GetMessage или PeekMessage. По умолчанию в очереди может находиться до восьми сообщений. Программы, которые требуют более емкой очереди сообщений, могут увеличить ее размер с помощью SetMessageQueue.

Aspect Ratio — Коэффициент масштабирования

Коэффициент масштабирования описывает соотношение между высотой пикселя и его шириной. Другая интерпретация этого термина трактует его как степень "квадратности" пикселей на экране. Дисплей CGA имеет коэффициент масштабирования 2/1, EGA - 1.33/1, VGA - 1/1.

Background Color — Цвет фона

Цвет фона - это атрибут контекста устройства, который используется для рисования текста, линий со стилем (несплошных) и кистей штриховки. Он контролируется установкой фонового режима, другого атрибута контекста устройства.

BitBlt — BIT BLock Transfer

Эта функция GDI копирует прямоугольную побитовую область из одного места в другое. BitBlt наиболее наглядна при перемещении окон по экрану дисплея. BitBlt также используется для быстрого появления и удаления меню. Прикладные программы часто используют функцию BitBlt для перемещения побитовых изображений на устройства вывода (дисплей или принтер). Программа может также скопировать изображение экрана дисплея в битовый образ.

Bitmap — Битовый образ

Битовый образ - одно из псевдоустройств (другим является метафайл), которое GDI использует для хранения изображений. Битовые образы используют ОЗУ для хранения прямоугольных изображений и создаются путем запроса GDI о выделении ОЗУ для хранения изображения. После такого выделения битовый образ представляет невидимую поверхность для рисования путем вызова любых программ рисования GDI. Битовые образы являются также типом ресурса, который позволяет программе хранить графические изображения в исполняемом файле. Третьим типом битовых образов являются зависящие от устройств битовые образы (DIB), которые предоставляют зависящий от устройства способ хранения графической информации.

Brush — Кисть

Объект рисования GDI, который используется для заполнения областей. Имеется три типа кистей: сплошная, по образцу и штриховая. Каждый контекст устройства содержит кисть, используемую для заполнения области при вызове программ заполнения фигур (Rectangle, Ellipse, Polygon и т.д.). Другая функция, использующая кисть, - это программа BitBlt для отмены операции BitBlt.

Call-back Functions — Функции обратного вызова

Функция обратного вызова дает Windows возможность связаться с программой, непосредственно вызвав ее подпрограмму.

Call thunk — Код вызова

Очень маленький фрагмент кода, который используется Windows в реальном режиме как мост с удаленными функциями в перемещаемом или удаляемом сегменте кода.

Caret — Каретка

Объект интерфейса пользователя, который служит в качестве указателя клавиатуры аналогично тому, как курсор служит указателем мыши. Каретка представляет собой мигающий битовый образ, который указывает пользователю фокус клавиатуры и организует обратную связь с текущей позицией окна.

Casting — Преобразование типа

Относится к способности компилятора языка C преобразовывать один тип данных в другой. В следующем примере переменной `i` типа `int` присваивается значение переменной `lValue` типа `long`:

```
long lValue;  
int i;  
lValue=1245;  
i=(int)lValue;
```

Преобразование типа отменяет установленные компилятором по умолчанию преобразования и отправку предупреждающих сообщений, которые компилятор обычно генерирует в подобных ситуациях. Программисты, работающие в Windows, могут опускать определенные типы преобразований, которые требовались в более старых версиях компилятора C. Например, в реакции на сообщение `WM_PAINT` все еще очень часто встречается конструкция:

```
PAINTSTRUCT ps;  
BeginPaint(hwnd, (LPPAINTSTRUCT)&ps);  
  
EndPaint(hwnd, (LPPAINTSTRUCT)&ps);
```

В данном случае нет никакой опасности, поэтому явное преобразование типа необязательно, и при этом определенные типы проблем скрываются. Например, если в предыдущем примере пользователь не укажет символ "&" благодаря преобразованию типа, компилятор не обнаружит ошибки:

```
PAINTSTRUCT ps;  
BeginPaint(hwnd, (LPPAINTSTRUCT)ps);
```

CDECL Calling Convention — Соглашения о вызове CDECL

Соглашения о вызове CDECL описывают способ передачи параметров в стек подпрограммы (справа налево) и обязанность вызывающей программы очистить параметры стека. Эти соглашения также позволяют определять программы с разным числом параметров, но имеют

Class Database — База данных классов	недостаток в том, что создаваемый код будет несколько больше и медленнее, чем альтернативное соглашение о вызове PASCAL.
Client Area Coordinates — Координаты "клиентной области"	Относится к набору классов окон, которые зарегистрированы в системе с помощью RegisterClass.
Clipping Region — Область прорисовки	Описывают систему координат с началом (0, 0) в левом верхнем углу окна. Единицы координат "клиентной" области совпадают с пикселями.
Clipping — Отсечение	Это замкнутая область, используемая для прорисовки. Внутри области допускается рисование. Вне ее рисование невозможно. Области прорисовки в Windows всегда описываются в терминах прямоугольников или групп прямоугольников.
Clipboard	Отсечение описывает поведение программ рисования GDI при распознавании некоторых заданных границ. Отсечение определяется в терминах замкнутых областей: внутри области прорисовки допускается рисование, вне ее не допускается.
Code Page — Страница кодов	Clipboard позволяет пользователю обмениваться данными между программами. Элементы меню Edit служат первичным интерфейсом пользователя с Clipboard и содержат опции Cut, Copy и Paste. Кроме того, для команд Clipboard определен стандартный набор клавиш акселераторов.
Common User Access (CUA) — Общий доступ пользователя	Страница кодов определяет набор символов, а также изображения символов. Например, страница кодов 437 представляет собой стандартный набор символов для версии IBM-PC в США.
Compiler Memory Model — Модель памяти компилятора	Относится к элементу архитектуры прикладных систем (SAA) фирмы IBM, который описывает стандарты, разработанные для интерфейса пользователя с приложением и системным программным обеспечением.
Coordinate Transformation — Преобразование координат	Относится к набору значений по умолчанию, который компилятор устанавливает для адресации кода и данных.
Cursor — Курсор	Преобразование координат относится к способу интерпретации координат рисования в среде графического вывода информации. Три основных типа преобразования координат: перенос, масштабирование и поворот. В Windows преобразование координат GDI ограничивается переносом и масштабированием.
Cursor — Курсор	Битовый образ, который перемещается в ответ на перемещение мыши.

**Debug Version of Windows —
Отладочная версия Windows**

Описывается Windows, которая работает со специальными копиями библиотек динамической компоновки KERNEL.EXE (KRNL286.EXE, KRNL386.EXE), USER.EXE и GDI.EXE.

**Default Window Procedure —
Стандартная процедура окна**

Библиотечная программа Windows, которая обрабатывает сообщения не из "клиентной" области: нажатие системных клавиш и другие не обрабатываемые процедурами окна сообщения.

Desktop Window — Настольное окно

Относится к окну, которое покрывает весь экран дисплея и располагается за любым другим окном системы.

**Device Capability Bits — Биты
возможностей устройства**

Относятся к набору флагов, которые выставляются драйверами дисплея GDI или принтера, и описывают физические возможности устройств. GDI использует эти флаги для определения того, нужно ли посылать запрос высокого уровня на рисование непосредственно драйверу устройства, либо моделировать этот запрос программно и направить драйверу устройства последовательность запросов рисования низкого уровня.

Device Context (DC) — Контекст устройства

Структура данных, созданная и поддерживаемая GDI для обеспечения аппаратно-независимых операций на дисплее, принтере, метафайле или битовом образе. Контекст устройства служит трем целям: содержит набор инструментальных средств рисования, организует связь с конкретным устройством и позволяет программе рисовать на этом устройстве.

Device Independent Bitmap (DIB) — Аппаратно-независимый битовый образ

Аппаратно-независимые битовые образы предоставляют стандартный формат для хранения цветной графической информации. Четыре формата DIB: 1 бит на пиксель (монохром), 4 бита на пиксель (16 цветов), 8 битов на пиксель (256 цветов) и 24 бита на пиксель (16 млн цветов).

Dialog Box — Блок диалога

Это окно, внутри которого есть другие окна, часто называемые управляющими элементами диалога. Блок диалога обычно используется для получения дополнительной информации от пользователя, нужной для выполнения некоторой команды.

Dialog Box Control — Управляющий элемент блока диалога

Управляющий элемент блока диалога - это окно, которое располагается в блоке диалога и обеспечивает проведение некоторых операций. Среди классов окон, которые определены для управления блоком диалога, можно назвать кнопки, комбинированные блоки, блоки редактирования, блоки списка, полосы прокрутки и статические блоки.

Dialog Box Coordinates — Координаты блока диалога

Координаты блока диалога предоставляют аппаратно-независимый способ задания расположения элементов блока диалога. Элементы блока диалога относятся к шрифту системы либо к любому другому определенному для этих целей шрифту.

Dialog Box Editor — Редактор блока диалога

Это графический инструмент размещения и разработки блока диалога и стиля его управлений. Фирма Microsoft поставляет редактор блока диалога как часть инструментальных средств разработки программного обеспечения (SDK).

Disabled Window — Запрещенное окно

Окно, которое не воспринимает ввод с клавиатуры или от мыши, например, запрещенное родительское окно модального блока диалога.

Discardable Memory — Освобождаемая память

Освобождаемый объект памяти можно удалить с помощью менеджера памяти, если иначе не удастся получить требуемый объем памяти. При определении сегмента, который будет высвобождаться, Windows использует алгоритм, определяющий наиболее редко используемый сегмент.

Drawing Attribute — Атрибут рисования

Сделанная установка или объект рисования в контексте устройства, который может сменить свое проявление при выводе различными программами рисования GDI. Примерами атрибутов рисования служат перья, кисти, шрифты, режимы распределения и цвета текста.

Dynamic Data Exchange (DDE) — Динамический обмен данными

Механизм обмена данными построен поверх механизма передачи сообщений Windows. Взаимодействие DDE называется диалогом. В диалоге DDE всегда участвуют два лица, один называется клиентом, а другой сервером. Есть несколько типов диалога DDE - обмен данными, отправка данных, исполнение команд и возвращение данных в базу данных сервера.

Dynamic Link Library — Библиотека динамической компоновки

Это файл, который содержит код или данные, совместно используемые различными прикладными программами в процессе исполнения. Например, ключевыми компонентами, составляющими саму Windows, являются наборы библиотек динамической компоновки KERNEL.EXE, USER.EXE и GDI.EXE наряду с набором драйверов. Шрифты являются примером библиотек динамической компоновки, которые не содержат кода, а только данные, совместно используемые различными программами.

Dynamic Linking — Динамическая компоновка

Это процесс, при котором различный код и данные различных модулей, прикладные программы и библиотеки динамической компоновки, соединяются друг с другом в процессе исполнения.

Event-driven — Управление событиями

Программное обеспечение, которое управляется событиями, построено таким образом, что оно обрабатывает внешние события, не обязательно происходящие в определенной последовательности. Традиционными примерами управляемого событиями программного обеспечения могут служить операционная система или драйвер устройства, которые реагируют на определенные коды прерывания. Но развитие персональных компьютеров и диалогового программного обеспечения привнесло эту концепцию в прикладное программирование. Графический интерфейс пользователя (GUI) создает среду программирования, существенно облегчающую создание прикладного программного обеспечения, управляемого событиями.

Expanded Memory Specification (EMS) — Спецификация расширенной памяти

Спецификация расширенной памяти относится к протоколу, который впервые был представлен в 1984 году фирмами Lotus, Intel и Microsoft как способ разрешения проблемы имеющегося ограничения в 640 Кбайт при работе в реальном режиме. Windows 1.x использует память EMS для буферизации остановленных приложений DOS. Windows 2.x обеспечивает расширенную поддержку со стороны EMS тем приложениям Windows, которые увеличивают адресуемое пространство способом, доступным каждому прикладному программисту. Windows 3.x продолжает линию поддержки EMS, хотя сама Windows использует EMS только в реальном режиме. В других режимах работы программа Windows может получать доступ к памяти EMS, но сама система не использует EMS, так как располагает более гибкими возможностями расширенной памяти другого типа.

Extended Memory — Дополнительная память

Понятие дополнительной памяти относится к памяти, которая обычно не доступна в реальном режиме, а требует операций в защищенном режиме. Дополнительная память допускает адресацию пространства компьютера, если объем памяти превышает один мегабайт в реальном режиме, и получение доступа до 16 Мбайт на процессоре 80286 и до 4 Гбайт на процессоре 80386.

Extended Memory Specification (XMS) — Спецификация дополнительной памяти

Спецификация дополнительной памяти определяет интерфейс для использования областей памяти свыше 640 Кбайт. Это включает использование верхних блоков памяти (UMB) от 640 до 1024 Кбайт, высших областей памяти (HMA) от 1024 до 1088 Кбайт и блоков расширенной памяти (EMB) свыше 1088 Кбайт.

File Manager — Менеджер файлов

Одно из средств, которые фирма Microsoft предоставляет с Windows 3.0.

Fixed Memory — Фиксированная память

Фиксированный объект памяти - это такой объект, логический адрес которого не меняется. Однако в защищенном режиме физический адрес фиксированного блока памяти может изменяться, если только память не была зафиксирована с помощью вызова GlobalFix. Кроме того, в режиме 386 Enhanced фиксированный объект памяти можно выгрузить на диск, если только он не заблокирован с помощью вызова GlobalPageLock.

Function Prototype — Прототип функции

Прототипы функций предоставляют компилятору языка C некоторые средства автоматического обнаружения ошибок, включая проверку корректности использования возвращаемого значения, корректного числа и типа параметров. Приведем пример прототипа функции из Window.h:

```
BOOL FAR PASCAL TextOut (HDC, int, int, LPSTR, int);
```

Global Descriptor Table (GDT) — Таблица глобальных дескрипторов;

Это область данных, которую процессор Intel 80286 и более поздние модели используют в защищенном режиме. Данная таблица позволяет процессору преобразовывать значения селектора сегмента в физические адреса сегментов. Intel разработала GDT для организации областей данных, которые могли бы совместно использовать все процессы. Однако Windows не использует GDT для управления памятью, а берет для этого другую область данных - таблицу локальных дескрипторов (LDT).

Global Heap — Глобальная куча

Глобальная куча относится ко всей памяти, доступной самой Windows, его приложениям и другим компонентам, например драйверам устройств.

Global Memory Object — Объект глобальной памяти

Объект, расположенный в глобальной куче. В терминах архитектуры памяти фирмы Intel это сегмент.

Granularity — Зернистость

В контексте выделения памяти зернистость относится к объемам приращения при действительном выделении памяти. Менеджер глобальной динамической области использует приращение в 32 байта. Менеджер локальной кучи использует приращение в 4 байта.

Graphic Device Interface (GDI) — Интерфейс графического устройства

Это аппаратно-независимая графическая библиотека вывода Windows.

Graphical User Interface (GUI) — Графический интерфейс пользователя

Относится к типу операционной системы или среды работы, которая отображает вывод на графический экран дисплея. Другая характеристика системы GUI состоит в том, что она управляется

событиями. Это значит, что необходима иная модель программирования, помимо традиционного последовательного программирования, которое обычно использовалось для командных систем. Примером системы GUI является Microsoft Windows. Другие примеры включают Apple Macintosh, OS/2 Presentation Manager, совместно разработанный фирмами IBM и Microsoft, и GEM фирмы Digital Research Corporation. К такого типа системам можно отнести и различные Windows-подобные разработки: Motif фирмы Open System Foundation, Open Look фирмы Sun Microsystems и DEC-Windows фирмы Digital Equipment Corporation.

Handle — Дескриптор

Это 16-битовое целое число без знака, которое идентифицирует объект. В большинстве случаев значение дескриптора известно лишь библиотеке подпрограмм, которая его создает. Программа может модифицировать объект, передав дескриптор в библиотеку подпрограмм, которая затем воздействует на объект от имени программы.

Hardware Event Queue — Очередь аппаратных событий

Это буфер, который обслуживается Windows, для хранения событий клавиатуры и мыши, которые ожидают обработки прикладной программой.

High Memory Area (HMA) — Высшая область памяти

Высшая область памяти - это 64 Кбайт, размещенные сразу же за мегабайтным адресом процессора 80286 или 80386. Фирма Intel разработала данное пространство как первые 64 Кбайт дополнительной памяти, которые не являются частью адресуемого пространства в реальном режиме. Однако драйвер XMS (спецификация расширенной памяти), например, HIMEM.SYS, способен организовать доступ процессора 80286 или более мощных в эту область данных.

Hit-Test Code — Код проверки попадания

Относится к коду, возвращаемому процедурой по умолчанию в ответ на сообщение WM_NCHITTEST, которое идентифицирует область окна, где расположена мышь. Код проверки попадания используется, чтобы помочь менеджеру окна установить корректный курсор мыши.

Hook — Захват

Это подпрограмма, которая вызывается механизмом обработки сообщений Windows и позволяет управлять и изменять траекторию передачи сообщений в системе. Захваты устанавливаются на уровне системы и, следовательно, нужно быть особенно осторожным при их использовании, так как можно нарушить нормальный ход выполнения других программ и существенно замедлить скорость работы системы в целом.

Hungarian naming — Венгерская Запись Имен

Соглашение относительно создания имен переменных и функций, которое дает быстрый способ составления достаточно коротких, но содержательных идентификаторов.

Huge Memory Object — Большие объекты памяти

Объекты памяти в глобальной динамической области, которые имеют размер более 64 Кбайт. Windows размещает большие объекты памяти, выделяя для них два сегмента или более. Для организации доступа ко второму или последующим сегментам необходимо привлечение арифметики больших указателей. Это включает изменение сегментной части адреса и части отклонения в адресе. Сегментная часть адреса изменяется путем смены значения сегмента с использованием значения `_AHINCR`. Это значение добавляется к адресу сегмента для получения доступа к следующему сегменту в цепи сегментов большого объекта.

Icon — Пиктограмма

Это графический символ, напоминающий пользователю о присутствии программы, файла или объекта данных, который в настоящий момент закрыт, но к которому можно организовать доступ в будущем.

Import Library — Библиотека импорта

Библиотека импорта дает компоновщику информацию об экспортных точках входа библиотеки динамической компоновки. Библиотека импорта позволяет компоновщику создавать запись размещения в файле `.EXE` программы, благодаря чему механизм динамической компоновки Windows обеспечивает вызов соответствующего кода во время исполнения программы.

Instance — Экземпляр

Экземпляр программы - это копия программы в памяти. Windows позволяет нескольким экземплярам одной программы работать одновременно. Каждый экземпляр имеет свой собственный частный сегмент данных, но совместно использует код и сегменты ресурсов со всеми другими экземплярами программы.

Instance Thunk — Фрагмент экземпляра

Небольшой код, который создается в базе данных задачи программы (TDB) для определения сегмента данных при вызове экспортируемых функций. Фрагмент экземпляра создается с помощью `MakeProcInstance` и освобождается с помощью `FreeProcInstance`. За исключением процедур окна, каждому вызову процедуры требуется свой фрагмент экземпляра. Это относится к процедуре блока диалога, процедурам перечисления и определенным процедурам подкласса. Для любой из этих процедур не нужен фрагмент экземпляра, если они размещены в сегменте кода библиотеки динамической компоновки.

Interrupt Descriptor Table (IDT) — Таблица дескрипторов прерываний

Таблица дескрипторов прерываний (IDT) используется процессорами 80286 или более поздними моделями в защищенном режиме для хранения указателей обработчиков прерываний.

Keyboard Accelerator — Акселератор клавиатуры

Акселератор клавиатуры предоставляет средства определения комбинаций нажатых клавиш, которые интерпретируются как команды. Акселераторы клавиатуры моделируют сообщения выбора вариантов меню и минимизируют специальную обработку, требуемую обычно для обеспечения ввода команд для программы с клавиатуры.

Keyboard Focus — Фокус клавиатуры

Фокус клавиатуры указывает Windows окно, которое будет принимать ввод с клавиатуры. Когда программа принимает фокус клавиатуры, она информируется об этом сообщением WM_SETFOCUS. При потере фокуса программа информируется об этом с помощью сообщения WM_KILLFOCUS.

Keyboard Scan Code — Сканкод клавиатуры

Это числовое значение, которое направляется аппаратурой клавиатуры в качестве извещения о том, что некоторая клавиша была нажата, отпущена или нажата в данный момент. Прикладные программы обычно не обрабатывают сканкоды, поскольку они представляют собой аппаратно-зависимые коды. В Windows сканкоды подвергаются двум трансляциям перед тем, как они появятся в программе в виде символов ASCII. Первая трансляция проводится из сканкода в код виртуальной клавиши, а вторая из кода виртуальной клавиши в код ASCII.

KERNEL — Ядро

Это один из трех ключевых компонентов Windows, который отвечает за управление памятью, динамическую компоновку, ресурсы, таблицы атомов, управление модулями, интерфейс с DOS и другие услуги операционной системы, которые предоставляются программам Windows.

Load On Call segment — Сегмент, загружаемый по вызову

Это код или сегмент ресурса, который загружается в память в момент востребования.

Local Descriptor Table (LDT) — Таблица локальных дескрипторов

Это область данных, используемая процессорами Intel 80286 или более поздними моделями в защищенном режиме. Таблица локальных дескрипторов позволяет процессору преобразовать значение селектора сегмента в физические адреса сегментов. Фирма Intel разработала LDT для обеспечения обработки частного адресного пространства. В Windows 3.0 одна LDT используется всеми прикладными программами. Однако в следующих версиях Windows LDT будут предоставляться каждой прикладной программе.

Local Heap — Локальная куча

Относится к динамической области, которая создается в сегменте данных модуля по умолчанию. При запуске приложения код запуска автоматически инициализирует локальную кучу прикладной программы с помощью вызова программы LocalInit. Библиотеки динамической компоновки, которым нужно использовать локальную кучу, должны сами явно вызвать LocalInit.

Logical Drawing Object — Логический объект рисования

Логический объект рисования - это описание GDI пера, кисти, шрифта или цвета. Благодаря этому удастся обеспечить аппаратно-независимый способ описания атрибутов рисования, установленных в контексте устройства.

Logical Font — Логический шрифт

Описание шрифта, которое компоновщик шрифтов GDI использует при выборе шрифта для рисования текста. Описание логического шрифта можно создать с использованием структуры данных LOGFONT.

Logical Pen - Логическое перо

Логическое перо описывает цвет, толщину и стиль линий, используемых программой.

Mapping Mode — Режим масштабирования

Это атрибут рисования контекста устройства, который описывает способ интерпретации координат рисования, передаваемых программам рисования GDI. Например, режим масштабирования по умолчанию MM_TEXT интерпретирует координаты как пиксели. Могут использоваться другие режимы масштабирования координат, интерпретирующие координаты как доли дюйма или сантиметра, или задаваться иной коэффициент масштабирования.

Marker — Маркер

Маркер - это графический примитив, который гарантированно отцентрирован в заданном месте. Хотя GDI сама непосредственно и не реализует маркеры, очень просто построить набор подпрограмм маркера, которые будут рисовать с помощью программ GDI (см. главу 7).

Message — Сообщение

Это 16-битовое значение без знака, которое информирует процедуру окна о некотором событии. Заранее определенные сообщения Windows идентифицируются в windows.h символьными константами, имена которых имеют префикс WM_. Например, сообщение WM_CREATE направляется к процедуре окна как извещение о том, что было создано окна данного класса.

Metafile — Метафайл

Псевдоустройство, которое может быть создано GDI для хранения графического изображения. Метафайл GDI - это структура данных, содержащая список вызовов программ GDI, которые нужно сделать для восстановления изображения, вместе с параметрами, обеспечивающими эти вызовы.

Module — Модуль

Особый тип объекта Windows, который отражает способ организации кода и данных на диске. Windows распознает два типа модулей: исполняемые программные файлы приложения (.EXE) и библиотеки динамической компоновки, которые могут иметь расширения .EXE, .DRV, .DLL или .FON.

Module Database — База данных модуля

Располагаемый в памяти образ заголовка файла прикладной программы или библиотеки динамической компоновки. Загрузчик Windows использует базу данных модуля для загрузки кода и ресурсов с диска по мере необходимости.

Module Definition File — Файл определения модуля

Текстовый файл ASCII содержит определение программы и информацию по использованию памяти. Файл определения модуля обычно имеет расширение .DEF. В момент создания программы файл определения модуля передается компоновщику для построения файла .EXE или .DLL.

Mouse Capture — Захват мыши

Обычно сообщения мыши посылаются к окну, расположенному под курсором мыши. Однако программа может ограничить поток сообщений мыши и направить его конкретному окну, осуществив захват мыши. Это делается с помощью вызова SetCapture. Для освобождения сообщений мыши программа вызывает ReleaseCapture.

Movable Memory — Перемещаемая память

Перемещаемые объекты Windows принадлежат не только реальному режиму Windows. В реальном режиме эти объекты перемещаются, только если они не захвачены программой. В защищенном режиме сегменты перемещаются в любой момент времени, поскольку их физические адреса скрыты от программы. В защищенном режиме аппаратура управления памятью преобразует логические адреса, которые использует прикладное программное обеспечение, в физические адреса с помощью таблицы дескрипторов. Менеджер памяти Windows может перемещать объекты в физической памяти без изменения их логических адресов. Это означает, что в защищенном режиме перемещение памяти остается прозрачным для прикладного программного обеспечения.

Multiple Document Interface (MDI) — Интерфейс множественного документа

MDI описывает стандарт интерфейса пользователя, при котором для каждого нового файла или документа, открываемого в программе, открывается новое окно.

Nonpreemptive Scheduling — Невытесняющее планирование

Описывает способ планирования работы программ в многозадачной системе. При невытесняющей системе планирования операционная система не прерывает выполнение программы.

OEM Character Set — Набор символов

Относится к набору символов для данной аппаратуры. Для IBM-совместимых компьютеров, изготовленных для продажи в США, это означает таблицу кодов 437.

Page Locking — Запирание страницы

Процесс, при котором память в режиме 386-Enhanced не выгружается на диск. Это выполняется с помощью вызова программы GlobalPageLock и применяется обычно для критичных по временным характеристикам драйверов устройств, которые должны резидентно располагаться в памяти.

Paragraph — Параграф

Когда процессор Intel 86 работает в реальном режиме, параграф - это наименьший фрагмент памяти, который можно выделить. Размер параграфа составляет 16 байт. Менеджер памяти Windows выделяет сегменты с приращением в два параграфа, что означает зернистость менеджера глобальной памяти в 32 байта.

Painbrush

Одна из программ, включенных в Windows 3.0.

Palette (PAL) — Палитра

Палитра служит двум целям: она описывает цвета, которые хранятся в аппаратно-независимом побитовом отображении (DIB), и позволяет программе запрашивать изменения в физической палитре дисплейного устройства, которое поддерживает работу с палитрой.

PASCAL Calling Convention — Соглашение о вызовах языка Pascal

Соглашение о вызовах языка Pascal описывает способ передачи параметров из стека в подпрограмму (слева направо) и возлагает на вызванную программу ответственность за удаление параметров из стека. Определенные по типу языка Pascal программы должны иметь фиксированное количество параметров, что приводит к несколько более короткому и быстрому коду по сравнению с альтернативным соглашением о вызовах CDECL.

Pen — Перо

Атрибут контекста устройства, описывающий цвет, стиль и толщину линий (см. Logical Pen).

Preemptive Scheduling — Вытесняющее планирование

Описывает способ планирования запуска задач в многозадачной системе. В системе планирования с выгрузкой операционная система прерывает выполнение одной программы для запуска другой.

Preload Segments — Заранее загруженные сегменты

Код сегмента ресурса, помеченный как заранее загружаемый, помещается в память до запуска программы на выполнение.

Private Window Class — Частный класс окна

Частный класс окна резервируется для одной программы. Противоположны глобальные классы окон, которые доступны любой программе системы.

Process Database (PDB) — База данных процесса

Созданная Windows структура данных, которая обеспечивает основанные на DOS данные процессов. Windows добавляет свои собственные элементы к этим структурам данных.

Program Manager — Менеджер программ

Относится к головному окну программы в Windows 3.0.

Protect mode — Защищенный режим

Относится к режиму работы процессоров Intel 80286 и более поздних моделей. Характеризуется средствами защиты схемы адресации, при которой программы не могут получить несанкционированного доступа к областям памяти, которыми они не владеют. При работе на процессоре 80286 в защищенном режиме допускается физическая адресация пространства до 16 Мбайт, а на 80386 и 80486 - до 4 Гбайт.

Pull-Model Processing — Вытягивающая обработка

Относится к стилю взаимодействия между операционной системой и прикладным программным обеспечением. При этой модели обработки прикладное программное обеспечение играет активную роль, оно сообщает операционной системе о возможном вводе. Это традиционный способ написания диалогового прикладного программного обеспечения, и именно Windows обрабатывает ввод с клавиатуры и мыши. Сообщения передаются процедуре окна с использованием процедуры PostMessage аналогичным образом, т.е. через процедуры GetMessage или PeekMessenger (см. также подталкивающую обработку).

Push-Model Processing — Подталкивающая обработка

Относится к стилю взаимодействия между операционной системой и прикладным программным обеспечением. При подталкивающей обработке активную роль играет операционная система, она вызывает подпрограммы прикладного программного обеспечения для выполнения нужных задач. Большинство неаппаратных сообщений Windows передаются таким образом. Например, когда для создания окна вызывается процедура CreateWindow, сообщение WM_CREATE направляется к процедуре вновь созданного окна. Сообщения передаются с помощью программы SendMessage, которая, в свою очередь, пересылает их на GetMessage и непосредственно вызывает процедуру окна.

Raster Operation — Растровая операция

Логическая операция или их комбинация, описывающая, как два или несколько вводов дают в результате заданный вывод. Один из типов растровых операций - коды ROP2, они являются атрибутом контекста устройства и описывают, как пиксели, линии и области комбинируются с изображаемой поверхностью. Другим примером

Return Thunks — Возвращаемый фрагмент

растровых операций могут служить коды ROP3, передаваемые в качестве параметров в функции BitBlt и PatBlt для описания того, как комбинируются исходные и целевые побитовые распределения с кистью.

Resource — Ресурс

Небольшой код, который используется Windows в реальном режиме для возвращаемого функцией значения, когда сегмент кода, содержащий вызов функции, удален из памяти.

Real Mode — Реальный режим

Объект данных, предназначенный только для чтения, который помещается в исполняемый файл в момент создания программы. С точки зрения управления памятью ресурс можно освободить и в случае необходимости повторно считать. С точки зрения интерфейса пользователя Windows ресурсы определяют блоки диалога, меню, курсоры, пиктограммы и битовые образы.

Scanner — Сканер

Относится к режиму работы семейства процессоров Intel 86. Реальный режим характеризуется адресуемым пространством в один мегабайт. Программы имеют доступ к реальным, физическим адресам памяти, вследствие чего данный режим и получил свое название.

Segment Selector — Селектор сегмента

Графический сканер считывает изображение с бумаги и преобразует его в цифровую форму, пригодную для создания битовых образов GDI.

Segmented Addressing — Сегментированная адресация

При работе семейства процессоров Intel 86 в защищенном режиме идентификатор сегмента называют селектором сегмента. Частью селектора сегмента является индекс таблицы дескрипторов LDT (локальная таблица дескрипторов) либо GDT (глобальная таблица дескрипторов), из которой считывается физическое местоположение сегмента памяти.

Software Development Kit (SDK) — Инструментальные средства разработки программного обеспечения

Семейство процессоров Intel 86 адресует память, используя схему сегментированной адресации. При сегментированной адресации требуется два элемента адреса памяти: идентификатор сегмента и смещение.

Software Migration Kit (SMK) — Инструментальные средства переноса программного обеспечения

В Windows являются продуктом, который фирма Microsoft предоставляет для облегчения создания программ Windows разработчиками программного обеспечения.

Продукт, который фирма Microsoft предоставляет для облегчения переноса программ Windows в среду OS/2 версии 1.2 и выше.

Stack Patching — Отслеживание стека

Относится к процессу, при котором ядро Windows в реальном режиме модифицирует ссылки стека на сегменты кода, которые перемещены или освобождены.

Standard Mode — Стандартный режим

Относится к режиму работы Windows, при котором разрешено использование защищенного режима адресации, что позволяет Windows использовать преимущества семейства процессоров фирмы Intel модели 80286 и выше.

Swap Kernel — Выгружаемое ядро

Специальная версия ядра Windows, которая работает только в реальном режиме и предоставляет информацию о загруженных и освобожденных сегментах программы. Выгружаемое ядро можно использовать для точной настройки набора сегментов программы для минимизации необходимой памяти.

Systems Application Architecture (SAA) — Архитектура прикладных систем

Созданный фирмой IBM набор стандартов для обеспечения преемственности программного обеспечения на персональных, мини- и больших ЭВМ. Относится к таким аспектам прикладного программного обеспечения, как интерфейс пользователя и программный интерфейс приложения (API). Стандарты интерфейса пользователя описаны в стандарте, который является частью SAA и носит название Общего интерфейса пользователя (CUA).

Task Database (TDB) — База данных задачи

Объект памяти, созданный планировщиком Windows для отслеживания принадлежащей задаче собственности. Эта собственность включает в себя текущий диск и каталог MS DOS, информацию о частных прерываниях задачи и указатель на базу данных программы DOS, известен также как префикс сегмента программы.

Text Alignment — Выравнивание текста

Атрибут контекста устройства, который описывает размещение строки текста относительно контрольной точки.

Text Color — Цвет текста

Атрибут контекста устройства, который описывает цвет изображаемого текста. Для текста используются только чистые, несмешанные цвета, таким образом GDI выставляет запрошенный цвет максимально близко к имеющемуся цвету устройства.

Thunk — Фрагмент

Небольшой код, который служит для динамической компоновки кода (в реальном режиме Windows).

Top-Level Window — Окно верхнего уровня

Перекрываемое (WS_OVERLAP) или всплывающее окно (WS_POPUP), которое не имеет родителя. Все окна верхнего уровня перечислены в списке задач Windows, который можно увидеть, если нажать комбинацию клавиш Ctrl+Esc.

Unexpected Application Error (UAE) — Неожиданная ошибка приложения

Ошибка, которая вызывает прекращение выполнения приложения. Среди причин появления неожиданных ошибок приложения могут быть ошибка общей защиты, неожиданная ошибка выгрузки или неожиданное прерывание.

Viewport Origin — Начало координат

Начало координат - это атрибут контекста устройства, который определяет преобразование координат после масштабирования при различных режимах распределения GDI.

Viewport Extent — Диапазон

Атрибут контекста устройства, который содержит пару значений X и Y, используемых для определения соотношений при изотропных и анизотропных режимах распределения GDI.

Virtual Key Code — Виртуальный код клавиши

Значение, которое представляет предварительную информацию о вводе с клавиатуры. Например, нет различия при вводе прописных и строчных букв и не принимается во внимание нажатие клавиш регистра Ctrl, Shift и Alt. Виртуальный код клавиши также не принимает во внимание состояние переключаемых клавиш Num Lock, Caps Lock или Scroll Lock. Виртуальный код клавиши - это промежуточный этап между скан-кодом (который зависит от аппаратуры клавиатуры) и аппаратно-независимым представлением символа ASCII.

Window Class — Класс окна

Заготовка для создания окон. Классы окон создаются с помощью вызова RegisterClasss и передачи указателя на структуру WNDCLASS с определением класса.

Window Extent — Диапазон окна

Диапазон окна - это атрибут контекста, который содержит пару значений X и Y, используемых для определения соотношений при изотропных и анизотропных режимах распределения GDI.

Window Origin — Начало координат окна

Атрибут контекста устройства, который определяет преобразование координат в общем координатном пространстве до масштабирования при различных режимах распределения GDI.

Приложение G. Установка интегрированной среды разработчика

Вы можете создавать файлы для всех образцов программ в этой книге. Это очень удобно, если Вы постоянно используете утилиту MAKE при создании программных продуктов. С другой стороны, лишь немногие программисты предпочитают развитую интегрированную среду фирмы Borland - Borland's Integrated Development Enviroment (IDE). Borland предоставляет как DOS-ориентированные, так и Windows-ориентированные версии IDE. В этом приложении рассматриваются установки, необходимые при создании программы Windows/OWL.

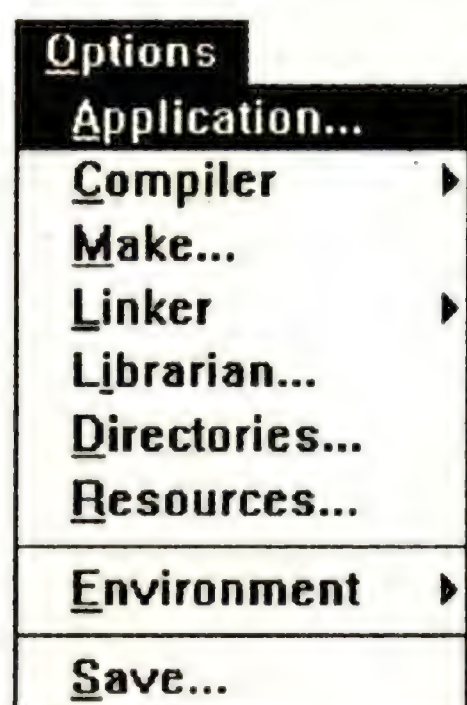


Рис. G.1. Элементы меню Option в Windows-ориентированной интегрированной среде разработчика.

DOS-ориентированная и Windows-ориентированная IDE имеют похожий пользовательский интерфейс. В этом приложении мы представляем меню и блоки диалогов в Windows-ориентированной IDE. Легко найти аналогичные меню и блоки диалогов в DOS-ориентированной IDE.

Запуск подразумевает выбор элемента в меню Option. На рис. G.1 представлено это меню со всеми своими подменю. Из 24 элементов меню, образующих блок диалогов, семь элементов содержат установки, которые являются решающими для успешного создания программы C++ Windows/OWL. Ниже приведены элементы меню, которые составляют необходимые блоки диалогов для установки развитой интегрированной среды:

- Options - Compiler - Code Generation...
- Options - Compiler - Advanced Code Generation...
- Options - Compiler - Entry/Exit Code...
- Options - Make...
- Options - Linker - Setting...
- Options - Linker - Libraries...
- Options - Directories...

На следующих страницах мы рассмотрим элементы каждого блока диалога, имеющие важное значение и помогающие сделать запуск быстрым и легким. Рисунки были взяты из ранней бета-версии Windows-ориентированной IDE. Хотя мы рассчитываем на то, что окончательная версия будет идентичной бета-версии, не удивляйтесь, если один или два элемента изменились.

Options - Compiler - Code Generation...

Model

Модель (или модель памяти) описывает метод доступа к кодам и данным по умолчанию. Модель малой памяти предполагает ближайшие ссылки на коды и данные. Хотя это и обеспечивает наилучшее выполнение, но ограничивает общее количество кодов и данных, доступных одной программе. С другой стороны, модель большой памяти предполагает дальние ссылки на коды и данные. Эта модель предпочтительна для многих программистов DOS и необходима при создании программ OWL, использующих динамическую библиотеку компоновки OWL.DLL.

Есть несколько ограничений, на которые рекомендуем обратить Ваше внимание. Если Вы создаете слишком много статических данных, которые предполагаете разместить в одном сегменте данных, то Ваша исполняемая программа будет содержать больше одного сегмента данных. Когда программа, исполняемая в Windows, имеет больше одного сегмента автоматических данных, Windows позволяет выполнять только единственный экземпляр. Если Вы попытаетесь запустить вторую копию программы, то получите сообщение об ошибке.

Assume SS equals DS

Условие равенства SS и DS. В этом блоке диалога определяется, каким способом компилятор работает с двумя регистрами центрального процессора - SS (регистром сегмента стека) и DS (регистром сегмента данных). Когда Вы создаете прикладную программу Windows, SS всегда должен быть равен DS. Однако при создании динамической библиотеки компоновки Windows, SS никогда не равен DS.

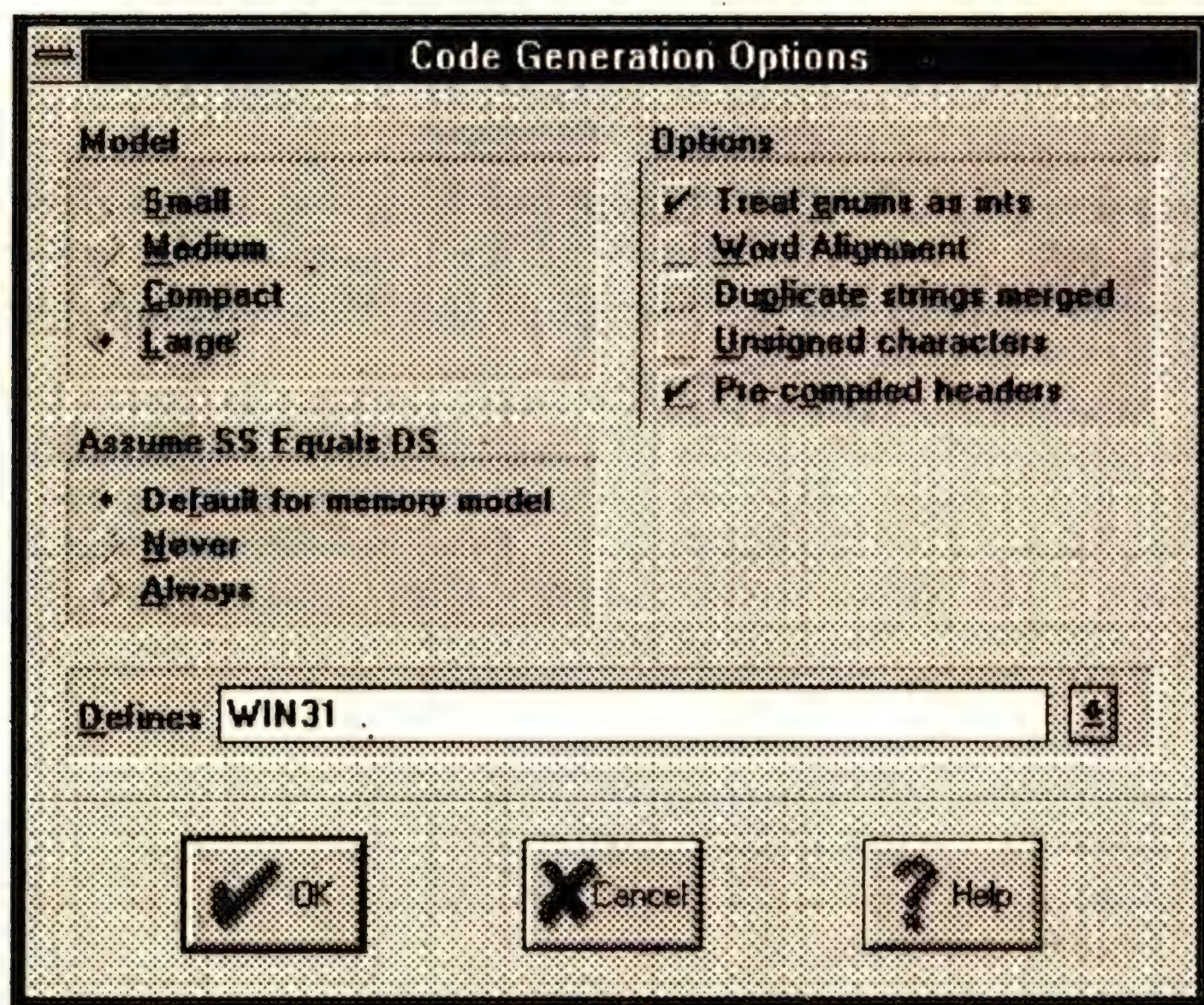


Рис. G.2. Диалоговое меню Compiler Code Generation.

Options: Precompiled headers

Выбор этой опции не обязателен, но он полезен для сокращения времени компиляции. При включении этой опции компилятор сохраняет двоичную копию объединенных файлов с расширением .h в файле с расширением .SYM.

Options - Compiler - Advanced Code Generation...

Из четырех элементов меню, которые мы будем обсуждать, только первый обязателен. Остальные три полезны для отладки и доступа к программе просмотра класса C++.

Generate underbars

Генерация подзаголовков. Вам необходимо выбирать эту опцию для доступа к процедурам библиотеки C (таким, как `strcpy`, `memcpy` и т.д.). В противном случае компоновщик не способен реализовать ссылки на эти процедуры.

Line numbers debug info

Отладочная информация о количестве строк. Выбор этой опции заставляет компилятор включать информацию о количестве строк исходной программы в файл .OBJ и в символьный файл .MAP. Это позволяет отладчику, такому, как Turbo Debugger для Windows, отслеживать исходный текст Вашей прикладной программы.

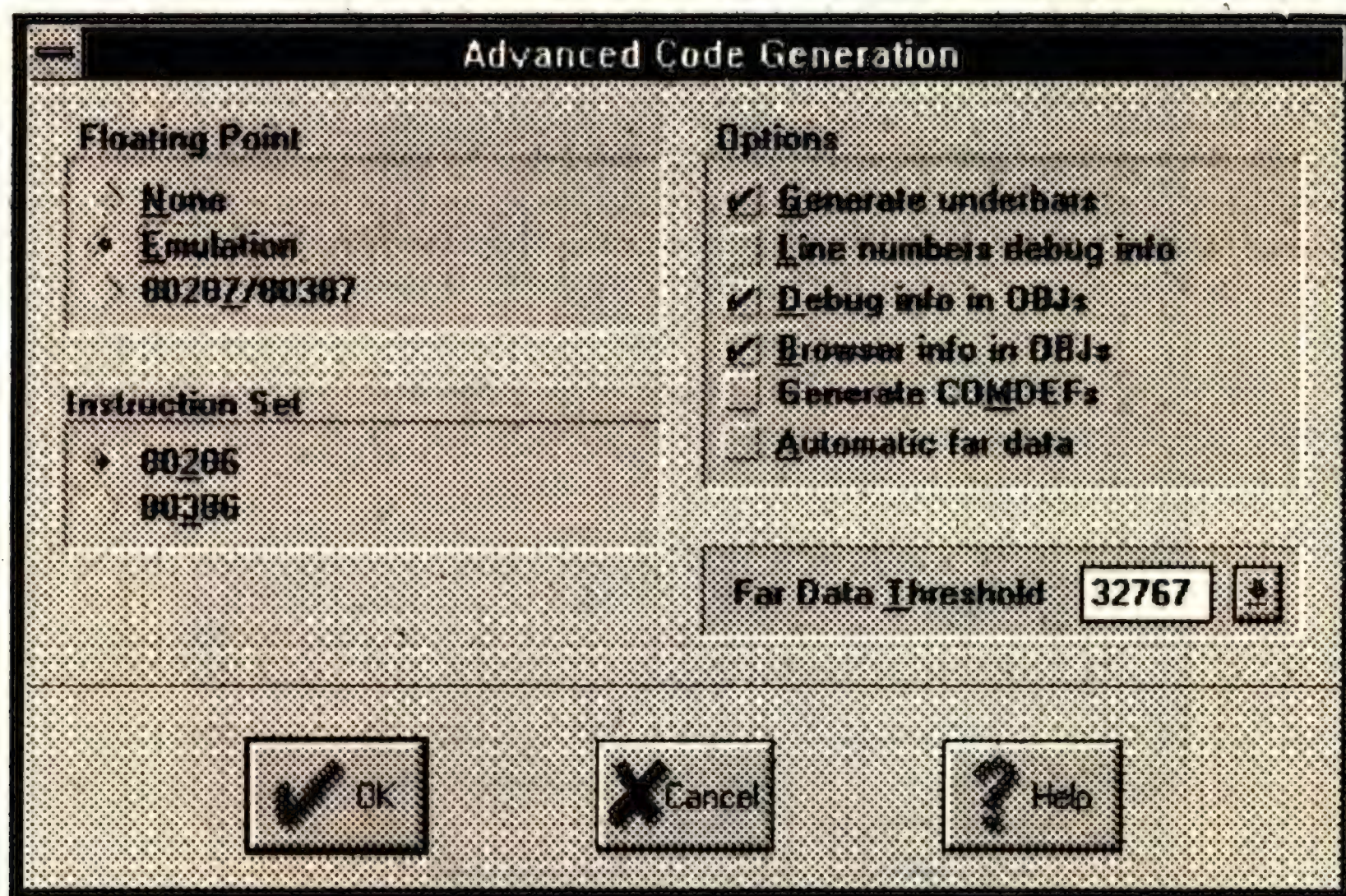


Рис. G.3. Диалоговое меню Compiler Advanced Code Generation.

Debug info in OBJs

Информация об отладке в объектных модулях. Выбор этого элемента обуславливает запись полной отладочной информации в файл .OBJ. Это необходимо, если Вы хотите использовать Turbo Debugger для Windows и сделать доступными имена локальных и глобальных переменных, а также имена функций.

Browser info in OBJs

Информация программы просмотра в объектных модулях. Установка этой опции позволяет программе просмотра класса C++ просматривать классы Ваших программ.

Options - Compiler - Entry/Exit Code...

Prolog/Epilog Code Generation

В начале каждой функции есть маленькая часть, которая называется прологом. Первоначальной работой пролога является установка стека, позволяющего функции получить доступ к параметрам и отвести место под локальные (автоматические) переменные (см. главу 17). Программу Windows можно вызвать с помощью библиотек Windows. Подобные функции "преддверия" специально обрабатываются для проверки правильности установки регистра сегмента данных (см. главу 19). Такие функции обычно экспортируемы, т.е. загрузчик Windows будет отлаживать пролог при загрузке программного сегмента в память. В программе Windows потребность функций "преддверия" может регулироваться одним из трех следующих способов.

* Windows all functions exportable (все функции Windows экспортируемы). Когда все функции экспортируемы, загрузчик Windows отлаживает каждую функцию, чтобы обеспечить их доступность для внешнего вызова. К сожалению, это наименее эффективный путь решения этой проблемы.

* Windows explicit functions exported (экспортирование явно заданных функций Windows). Эта установка влияет только на функции, которые имеют ключевое слово `_export`. Такая установка лучшая для использования, поскольку она самая эффективная. Однако при этом требуется, чтобы Вы использовали ключевое слово `_export` всегда, когда создаете функцию с обратным вызовом и передаете ее в Windows. При создании объектов окон или объектов диалога с использованием OWL Вам не нужно это ключевое слово, так как библиотеки OWL сами позаботятся о нем.

* Ускорение обратных вызовов Windows. Эта установка обеспечивает допустимый компромисс между двумя предыдущими. Она использует преимущество того, что в программе Windows сегмент стека (SS) обычно такой же, как и сегмент данных. Для всех дальних функций пролог генерируется, следствием чего регистр сегмента стека копируется в регистр сегмента данных (DS).

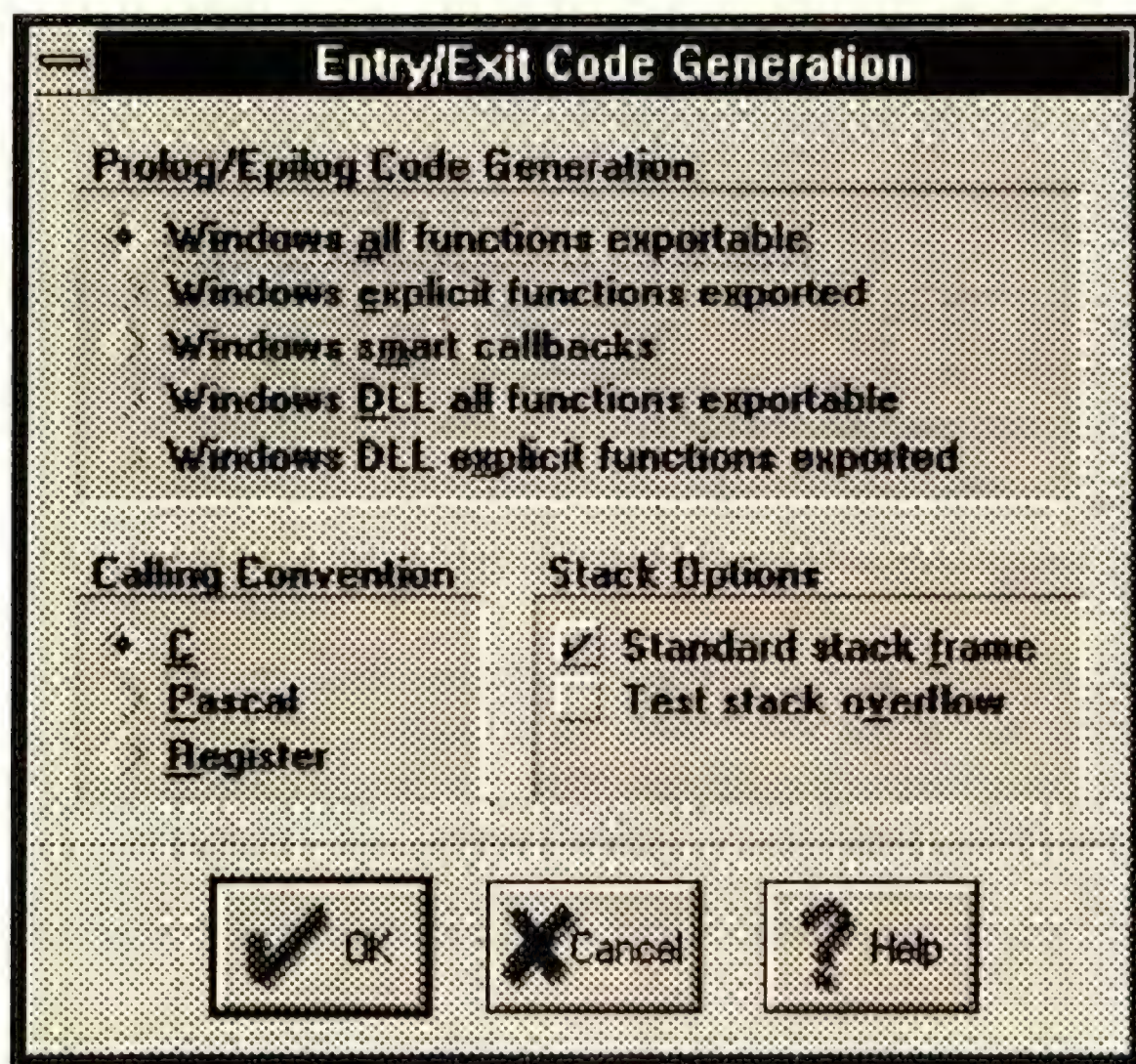


Рис. G.4. Диалоговое меню компилятора Entry Exit Code Generation.

Когда Вы создаете динамическую библиотеку компоновки, то должны выбрать одну или две установки из нижеследующих:

* Windows DLL all functions exportable (все функции Windows DLL экспортируемы). Хотя это и не самый эффективный путь, он позволяет вызвать из прикладной

программы любую дальнюю функцию в DLL. Дело в том, что каждая функция в DLL представляется соответствующим прологом.

* Windows DLL explicit functions exportable (все явно заданные функции Windows DLL экспортируемы)! Эта установка требует, чтобы Вы использовали ключевое слово `_export` для каждой функции, которая будет вызываться из прикладной программы. Это создает пролог в начале функции, который соответствующим образом устанавливает регистр DS для определения указателя на сегмент данных DLL.

Calling Convention

Соглашение вызова. От этого переключателя зависит установка по умолчанию, определяющая, как будет осуществляться вызов функции. Наиболее надежная для использования установка - это соглашение вызова C, так как программы, которые используют соглашение вызова Pascal, обычно имеют ключевое слово `pascal`.

Options - Make...

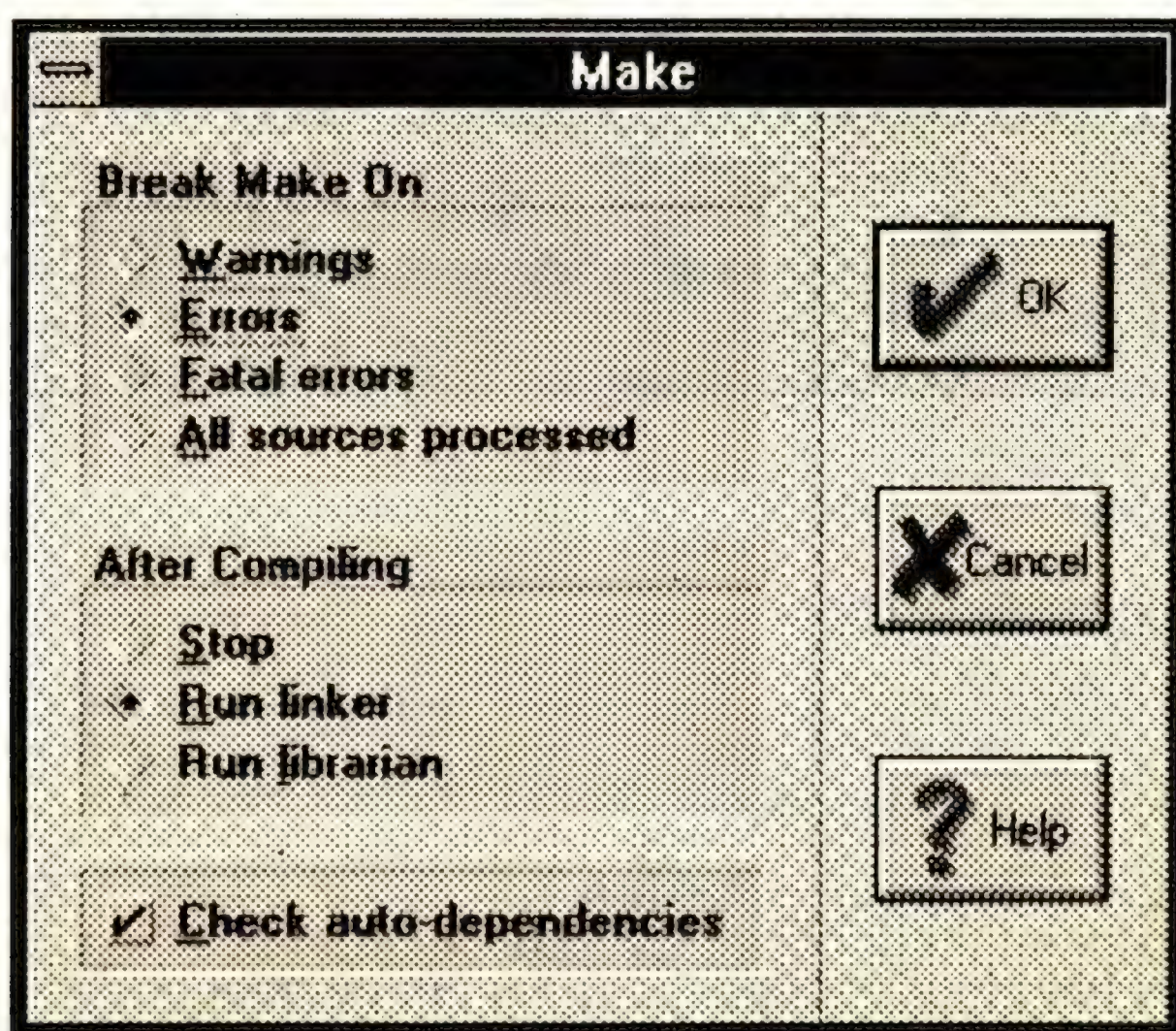


Рис. G.5. Диалоговое меню Make.

Check auto-dependencies

Проверка автоматических зависимостей. Вам необходимо отметить эту опцию, чтобы MAKE прочитал информацию о зависимостях из файлов с расширением `.h`.

Options - Linker - Setting...

Output

Выходные данные. Обязательно проверьте соответствующую опцию для выходного файла. Фактически единственной разницей между файлом прикладной программы (`.EXE`) Windows и файлом динамической библиотеки компоновки (`.DLL`) является один бит. Однако для достижения ожидаемых результатов этот бит должен быть установлен правильно.

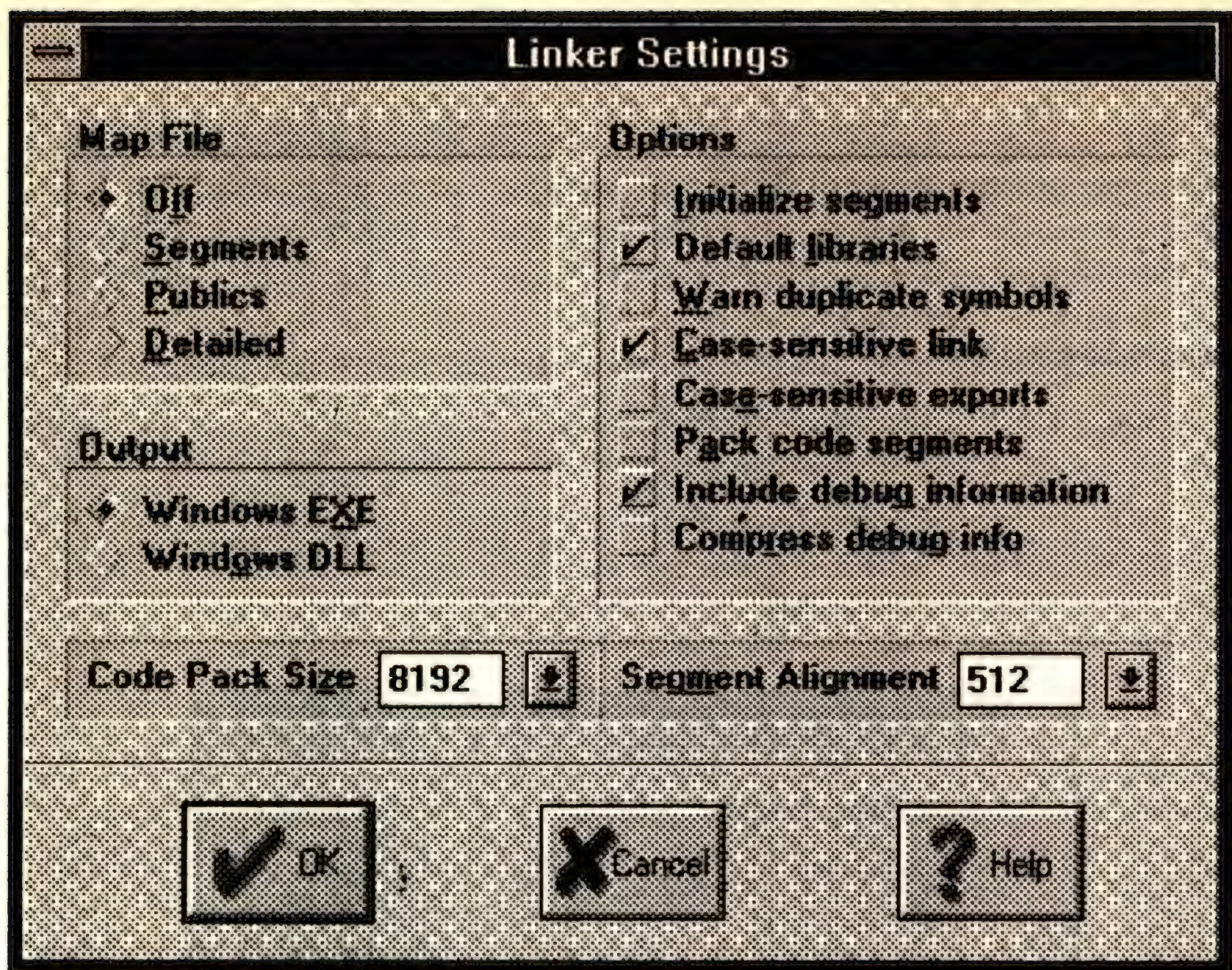


Рис. G.6. Диалоговое меню Linker Setting.

Options: Case-sensitive link

Компоновка с учетом регистра (различное написание заглавных и строчных букв). Поскольку языки C и C++ являются языками с учетом регистра, обязательно проверьте установку этого элемента.

Options: Include debug information

Включить информацию об отладке. Если Вы собираетесь использовать отладчик, подобный Turbo Debugger для Windows, обязательно проверьте эту опцию.

Options - Linker - Libraries...

Object Windows Library

Объектная библиотека Windows. Вы можете осуществить компоновку с библиотекой OWL одним из трех способов:

* None. Если выбран этот элемент, то Вы должны явно указать имя библиотеки OWL в Вашем проекте IDE.

* Static - Статический. Выбор этого элемента приводит к тому, что функции OWL должны располагаться в пределах исполняемого файла (.EXE) Ваших программ. Это требует различных файлов статической библиотеки компоновки OWL. Например, файл OWLWL.LIB содержит большую модель статической библиотеки компоновки OWL.

* Dynamic - Динамический. Если выбрать этот элемент, то функции OWL будут доступны из динамической библиотеки компоновки (OWL.DLL). Это требует динамической информационной библиотеки (OWL.LIB) во время компоновки программы. Заметьте, что эта опция недоступна всем компиляторам Borland C++.

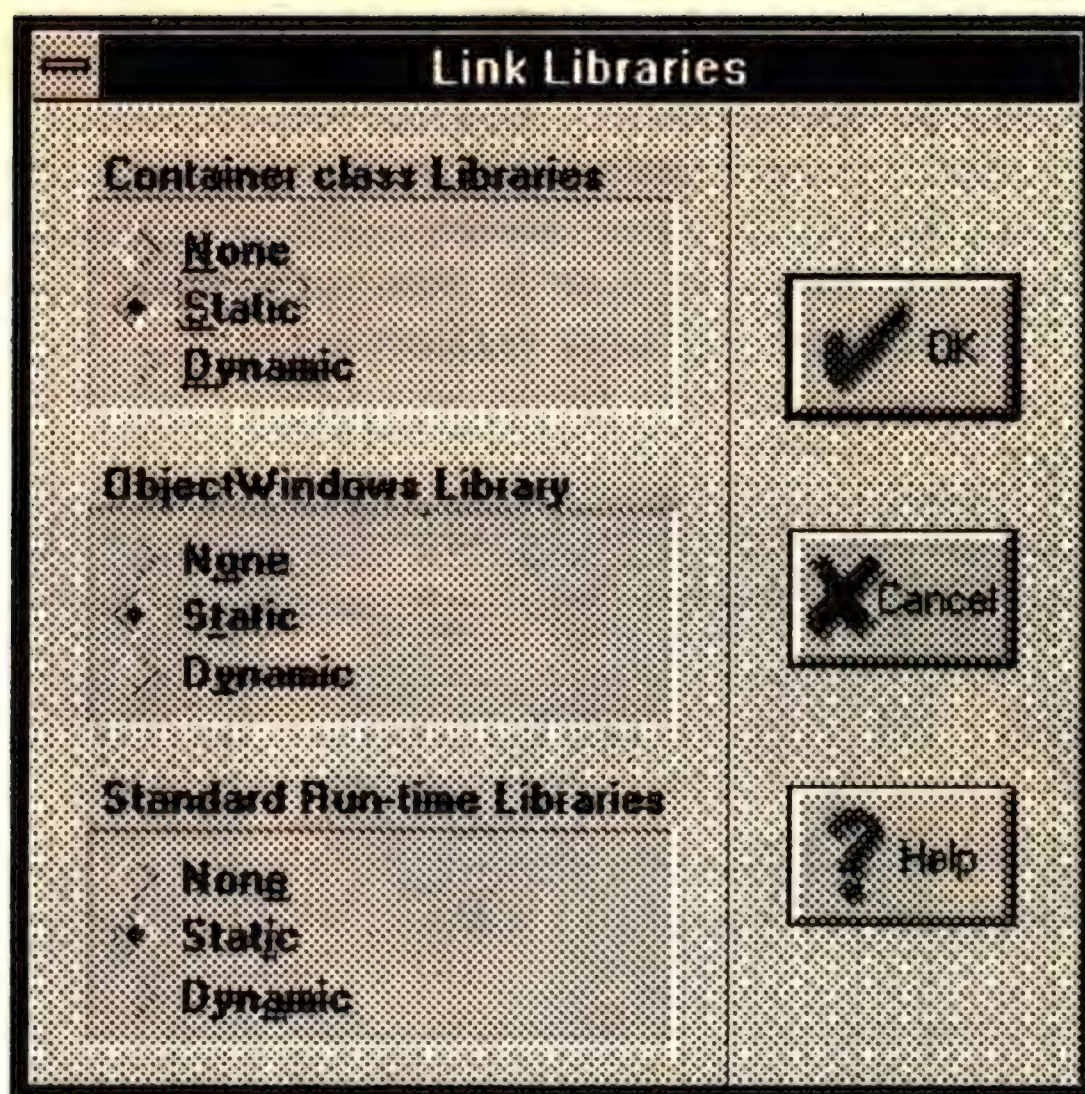


Рис. G.7. Диалоговое меню *Linker Libraries*.

Standard Run-time

Стандартная исполняющая система. Вы можете осуществить компоновку со стандартной библиотекой C/C++ исполняющей системы одним из трех способов:

* **None**. Можете выбрать этот элемент в том случае, если Вы хотите явно указать имя библиотеки процедуры C в списке проекта IDE.

* **Static** - Статический. Выбор этого элемента приводит к тому, что функции библиотеки процедур C должны быть расположены в пределах исполнительного файла (.EXE) Вашей программы.

* **Dynamic** - Динамический. Выбирайте этот элемент для того, чтобы использовать динамическую библиотеку компоновки как версию функций библиотеки процедур C (объединенных в файл BCRTL.DLL).

Options - Directories...

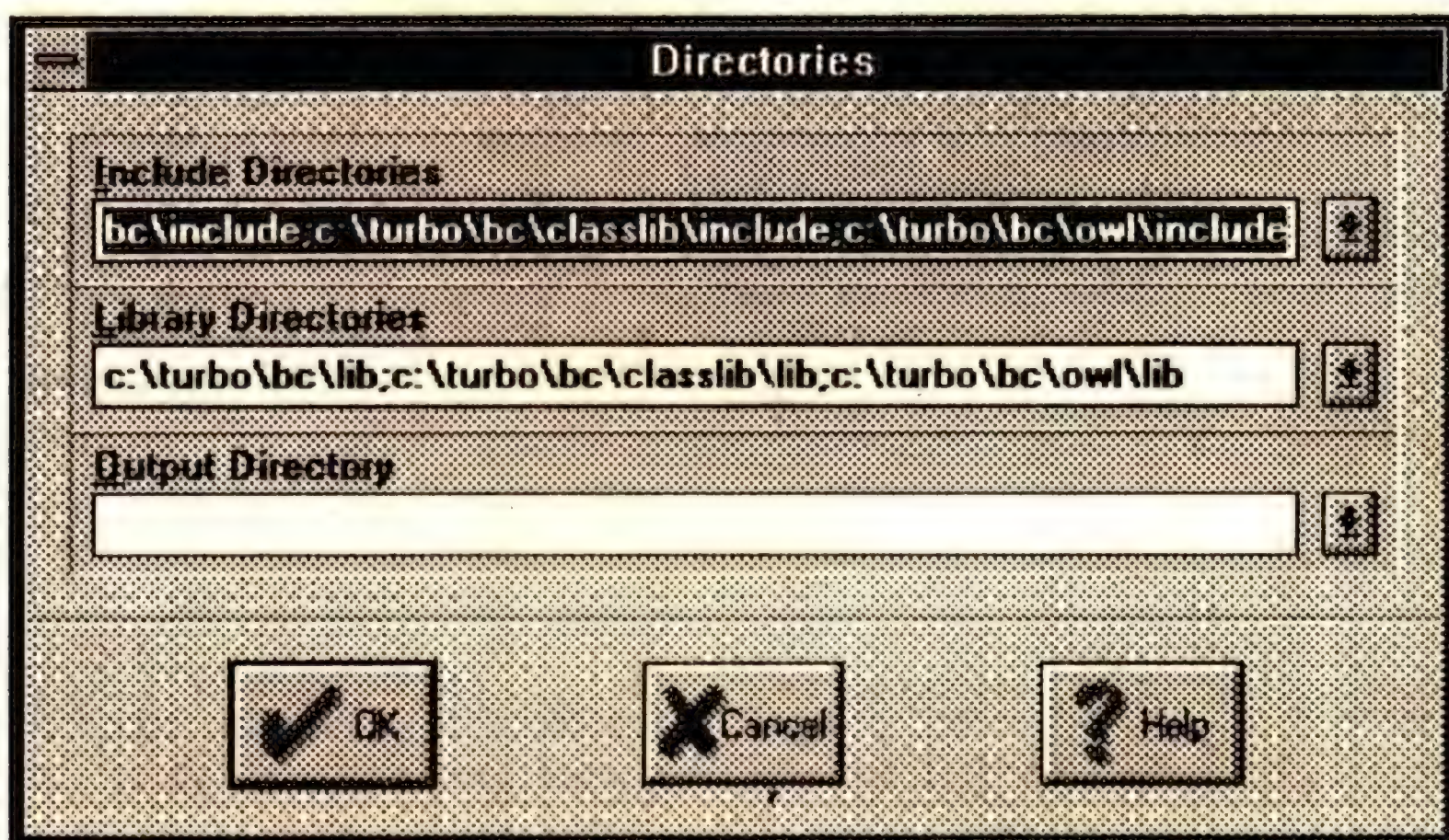


Рис. G.8. Диалоговое меню *Directories*.

Include Directories

Объединенные каталоги. Здесь следует перечислить каталоги, которые содержат объединенные файлы, необходимые компилятору. На этом рисунке представлены установки для компилятора Turbo C++, программа инициализации которых использует подкаталог \TCWIN. Если Вы используете компилятор Borland C++, то можете подставить \BORLAND для демонстрации установок.

Library Directories

Каталоги библиотек. Перечислите каталоги, в которых могут находиться библиотеки компоновщика. На этом рисунке показано расположение по умолчанию для компилятора Turbo C++.

Output Directory

Выходной каталог. Укажите каталог, в который должен быть записан исполняемый файл после успешной компиляции или компоновки.

Приложение Н. Программа MAGNIFY

Программа MAGNIFY используется в разных местах данной книги. Она существенно упрощает просмотр различных деталей изображения на экране подобно тому, как увеличительное стекло позволяет более подробно рассмотреть физические объекты. Коэффициент увеличения изображения можно регулировать с помощью линейки прокрутки. На рис.Н.1 проиллюстрирована работа программы MAGNIFY.

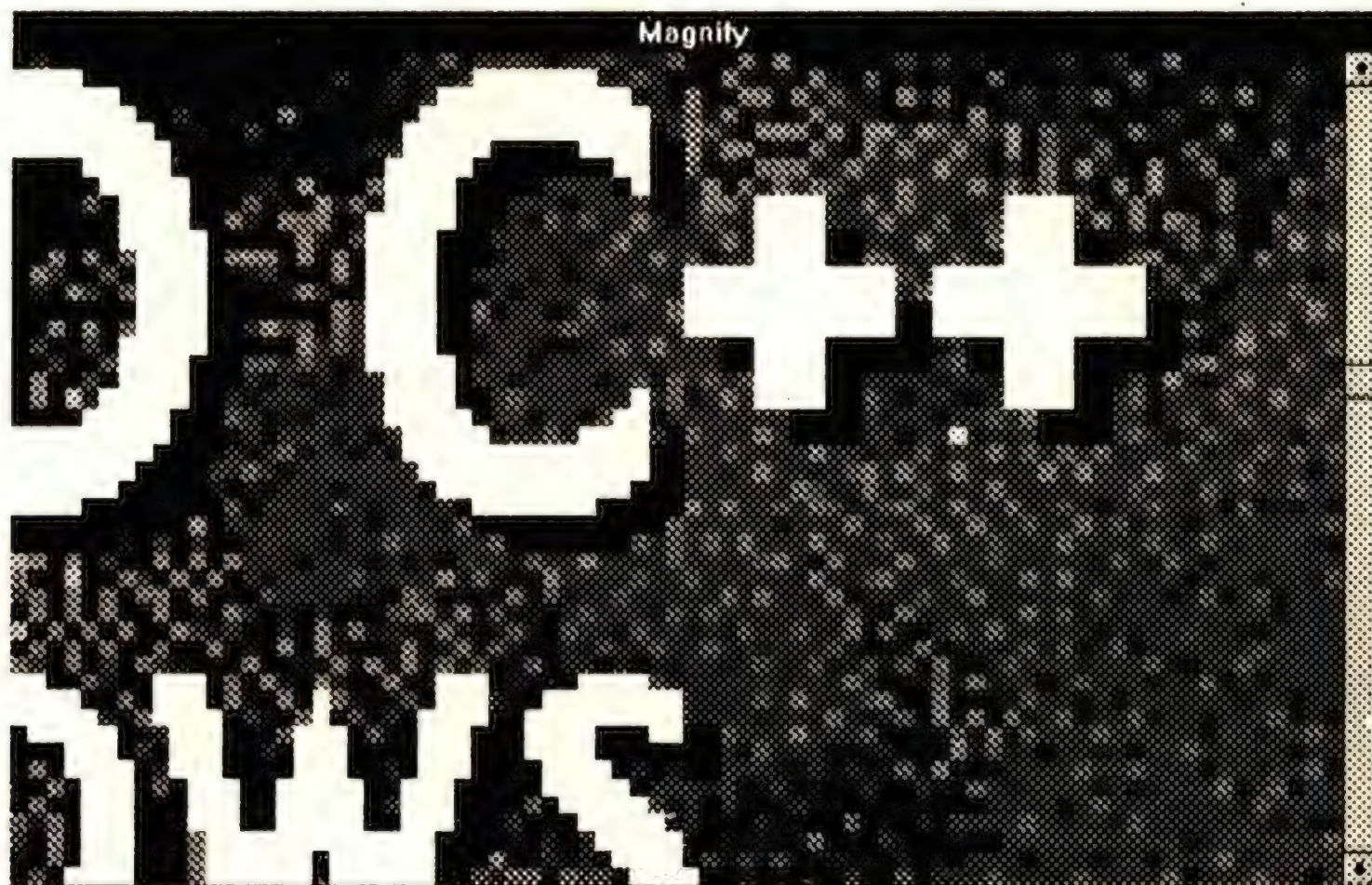


Рис.Н.1 Пример работы программы MAGNIFY

Ниже представлен исходный текст программы MAGNIFY:

MAKEFILE.MAK

.AUTODEPEND

Определения транслятора

INC=C:\BORLANDC\OWL\INCLUDE;C:\BORLANDC\CLASSLIB\INCLUDE;C:\BORLANDC\INCLUDE

CC = bcc -c -D_CLASSDLL -H -ml -WS -w -I\$(INC)

Неявные правила

.c.obj:

\$(CC) (\$<)

.cpp.obj:

\$(CC) (\$<)

Явные правила

Magnify.exe: Magnify.res Magnify.def Magnify.obj

tlink /c/C/n/P-/Twe/x @Magnify.LNK

RC Magnify.res Magnify.exe

Зависимости отдельных файлов

Magnify.obj: Magnify.cpp

Magnify.res: Magnify.rc Magnify.cur Magnify.ico

RC -R -FO Magnify.res Magnify.RC

MAKEFILE.LNK

```
c:\borlandc\lib\c0wl.obj+
Magnify.obj
Magnify,Magnify
\borlandc\owl\lib\owl.lib+
crtll.lib+
cwinl.lib+
import.lib+
mathl.lib+
cl.lib
Magnify.def
```

MAGNIFY.CPP

```
/*
MAGNIFY. - Растягивает кусок экрана
*/
```

```
#include <owl.h>
#define min(a,b) ((a < b) ? a : b)
#define max(a,b) ((a < b) ? b : a)
```

```
/*
Константы
*/
```

```
const int LINESIZE    = 1;
const int PAGESIZE    = 10;
const int SCROLLMIN   = 10;
const int SCROLLMAX   = 200;
```

```
/*
Описания классов
*/
```

```
class TMagnifyApplication : public TApplication
{
public:
    TMagnifyApplication (LPSTR lpszName,
                        HANDLE hInstance,
                        HANDLE hPrevInstance,
                        LPSTR lpszCmdLine, int nCmdShow);
    virtual void InitMainWindow();
};
```

```
class TNotRect
{
public:
    TNotRect (int X1, int Y1, int X2, int Y2);
    void Hide(HDC hdc);
    void Show(HDC hdc);
    void Move(HDC hdc, int X1, int Y1, int X2, int Y2);
private:
    BOOL bVisible;
    RECT rCurrent;
```



```

        void Invert(HDC hdc);
    };

class TMagnifyWindow : public TWindow
{
public:
    TMagnifyWindow (PTWindowsObject pwParent,
                    LPSTR lpszTitle,
                    PTModule pmModule);
    ~TMagnifyWindow ();
    virtual LPSTR GetClassName();
    virtual void GetWindowClass(WNDCLASS&);
    virtual void WMCreate (TMessage& Msg) = [WM_CREATE];
    virtual void WMLButtonDown (TMessage& Msg) =
        [WM_LBUTTONDOWN];
    virtual void WMMouseMove (TMessage& Msg) = [WM_MOUSEMOVE];
    virtual void WMLButtonUp (TMessage& Msg) = [WM_LBUTTONUP];
    virtual void WMVScroll (TMessage& Msg) = [WM_VSCROLL];
    virtual void WMPaint (TMessage& Msg) = [WM_PAINT];
    virtual void WMSize (TMessage& Msg) = [WM_SIZE];

protected:
    BOOL bCapture;
    HDC hdcScreen;
    HDC hdcWindow;
    int nStretch;
    POINT ptAnchor;
    POINT ptLast;
    RECT rClient;          // Размер клиентской области.
    RECT rSource;          // Координаты исходной области.
    TNotRect * pnrTrack;
    void ClientToSource(LPRECT lpIn, LPRECT lpOut);
};

```

```

/*
Функция WinMain
*/

```

```

int PASCAL WinMain (HANDLE hInstance,
                    HANDLE hPrevInstance,
                    LPSTR lpszCmdLine,
                    int nCmdShow)
{
    TMagnifyApplication Magnify ("Magnify",
                                  hInstance,
                                  hPrevInstance,
                                  lpszCmdLine,
                                  nCmdShow);

    Magnify.Run();
    return Magnify.Status;
}

```



```

/*
Компонентная функция класса прикладной программы
*/

```

```

TMagnifyApplication::TMagnifyApplication (LPSTR lpszName,
                                           HANDLE hInstance,
                                           HANDLE hPrevInstance,
                                           LPSTR lpszCmdLine,
                                           int nCmdShow)
    :TApplication (lpszName,
                  hInstance,
                  hPrevInstance,
                  lpszCmdLine,
                  nCmdShow)
{
    /* Здесь инициализируется прикладная программа. */
}

```

```

/*
Компонентная функция класса прикладной программы
*/

```

```

void TMagnifyApplication::InitMainWindow ()
{
    MainWindow = new TMagnifyWindow (NULL,
                                     "Magnify",
                                     NULL);
}

```

```

/*
Компонентная функция класса TMagnifyWindow
*/

```

```

TMagnifyWindow::TMagnifyWindow (PTWindowsObject pwParent,
                                LPSTR lpszTitle,
                                PTModule pmModule)
    :TWindow (pwParent, lpszTitle, pmModule)
{
    hdcWindow = 0;
    bCapture = FALSE;
    hdcScreen = CreateDC ("DISPLAY", 0, 0, 0);
    ptAnchor.x = 0;
    ptAnchor.y = 0;
    ptLast.x = 0;
    ptLast.y = 0;
    rClient.left = 0;
    rClient.top = 0;
    rSource.left = 0;
    rSource.top = 0;
    rSource.right = 1;
    rSource.bottom = 1;
    pnrTrack = 0;
    Attr.Style = WS_VSCROLL;
}

```



```

/*
Компонентная функция класса TMagnifyWindow
*/

```

```

TMagnifyWindow::~TMagnifyWindow ()
{
    DeleteDC (hdcScreen);
}

```

```

/*
Компонентная функция класса TMagnifyWindow
*/

```

```

LPSTR TMagnifyWindow::GetClassName ()
{
    return "Magnify:MAIN";
}

```

```

/*
Компонентная функция класса TMagnifyWindow
*/

```

```

void TMagnifyWindow::GetWindowClass (WNDCLASS& wc)
{
    TWindow::GetWindowClass (wc);
    wc.hIcon=LoadIcon (wc.hInstance, "snapshot");
    wc.hCursor=LoadCursor (wc.hInstance, "hand");
    wc.style = CS_HREDRAW | CS_VREDRAW;
}

```

```

/*
Компонентная функция класса TMagnifyWindow
*/

```

```

void TMagnifyWindow::WMCreate (TMessage& Msg)
{
    nStretch = SCROLLMIN;
    SetScrollRange (Msg.Receiver,SB_VERT,SCROLLMIN,SCROLLMAX,ALSE);
}

```

```

/*
Компонентная функция класса TMagnifyWindow
*/

```

```

void TMagnifyWindow::WMLButtonDown (TMessage& Msg)
{
    bCapture = TRUE;
    SetCapture (Msg.Receiver);
    hdcWindow = GetDC (Msg.Receiver);

    GetClientRect (Msg.Receiver, &rClient);
    ptAnchor = MAKEPOINT (Msg.LParam);
    ClientToSource (&rClient, &rSource);
    OffsetRect (&rSource, ptAnchor.x - (rClient.right/2),
                ptAnchor.y - (rClient.bottom/2));
    ClientToScreen (Msg.Receiver, (LPPOINT)&rSource.left);
}

```



```

ClientToScreen (Msg.Receiver, (LPPOINT)&rSource.right);
pnrTrack = new TNotRect (rSource.left, rSource.top,
                        rSource.right, rSource.bottom);
pnrTrack->Show (hdcScreen);
}

```

/*

Компонентная функция класса TMagnifyWindow

*/

```

void TMagnifyWindow::WMLButtonUp (TMessage& Msg)
{
    if (!bCapture) return;
    bCapture = FALSE;
    ReleaseCapture();
    ReleaseDC (Msg.Receiver, hdcWindow);
    ptLast = MAKEPOINT (Msg.LParam);
    ClientToScreen (Msg.Receiver, &ptLast);
    pnrTrack->Hide(hdcScreen);
    delete pnrTrack;
    pnrTrack = 0;
}

```

/*

Компонентная функция класса TMagnifyWindow

*/

```

void TMagnifyWindow::WMMouseMove (TMessage& Msg)
{
    POINT ptScreen;

    if (!bCapture) return;
    ptScreen = MAKEPOINT (Msg.LParam);
    ClientToScreen (Msg.Receiver, &ptScreen);
    CopyRect (&rSource, &rClient);
    rSource.left      = ptScreen.x - ptAnchor.x;
    rSource.top       = ptScreen.y - ptAnchor.y;
    rSource.right     = ptScreen.x + (rClient.right - ptAnchor.x);
    rSource.bottom    = ptScreen.y + (rClient.bottom - ptAnchor.y);
    ClientToSource (&rClient, &rSource);

    rSource.left      -= rClient.right / 2;
    rSource.top       -= rClient.bottom / 2;
    rSource.right     -= rClient.right / 2;
    rSource.bottom    -= rClient.bottom / 2;
    OffsetRect (&rSource, ptScreen.x, ptScreen.y);
    pnrTrack->Hide (hdcScreen);
    StretchBlt(hdcWindow, 0, 0, rClient.right, rClient.bottom,
              hdcScreen, rSource.left, rSource.top,
              rSource.right - rSource.left + 1,
              rSource.bottom - rSource.top + 1, SRCCOPY);

    pnrTrack->Show (hdcScreen);
    pnrTrack->Move (hdcScreen,
                  rSource.left,
                  rSource.top,

```



```
        rSource.right,  
        rSource.bottom);
```

```
/*  
Компонентная функция класса TMagnifyWindow  
*/
```

```
void TMagnifyWindow::WMPaint (TMessage& Msg)  
{  
    PAINTSTRUCT ps;  
  
    BeginPaint (Msg.Receiver, &ps);  
    StretchBlt(ps.hdc, 0, 0, rClient.right, rClient.bottom,  
        hdcScreen, rSource.left, rSource.top,  
        rSource.right - rSource.left + 1,  
        rSource.bottom - rSource.top + 1, SRCCOPY);  
    EndPaint (Msg.Receiver, &ps);  
}
```

```
/*  
Компонентная функция класса TMagnifyWindow  
*/
```

```
void TMagnifyWindow::WMSize (TMessage& Msg)  
{  
    rClient.right = Msg.LP.Lo;  
    rClient.bottom = Msg.LP.Hi;  
}
```

```
/*  
Компонентная функция класса TMagnifyWindow  
*/
```

```
void TMagnifyWindow::WMVScroll (TMessage& Msg)  
{  
    BOOL fScroll;  
  
    fScroll = FALSE;  
    switch (Msg.WParam)  
    {  
        case SB_LINEUP:  
            nStretch -= LINE_SIZE;  
            fScroll = TRUE;  
            break;  
        case SB_LINEDOWN:  
            nStretch += LINE_SIZE;  
            fScroll = TRUE;  
            break;  
        case SB_PAGEUP:  
            nStretch -= PAGE_SIZE;  
            fScroll = TRUE;  
            break;  
        case SB_PAGEDOWN:  
            nStretch += PAGE_SIZE;  
            fScroll = TRUE;  
    }
```



```

        break;
    case SB_THUMBTRACK:
        nStretch = Msg.LP.Lo;
        fScroll = TRUE;
        if (pnrTrack == 0)
        {
            pnrTrack = new TNotRect (rSource.left-1,
                                     rSource.top-1,
                                     rSource.right+1,
                                     rSource.bottom+1);
            pnrTrack->Show (hdcScreen);
        }
        break;
    case SB_ENDSCROLL:
        if (pnrTrack)
        {
            pnrTrack->Hide (hdcScreen);
            delete pnrTrack;
            pnrTrack = 0;
        }
        break;
    }

    if (fScroll)
    {
        nStretch = min (SCROLLMAX, nStretch);
        nStretch = max (SCROLLMIN, nStretch);

        SetScrollPos (Msg.Receiver, SB_VERT, nStretch, TRUE);
        ClientToSource (&rClient, &rSource);
        OffsetRect (&rSource, ptLast.x - (rClient.right/2),
                   ptLast.y - (rClient.bottom/2));

        if (pnrTrack)
        {
            pnrTrack -> Move (hdcScreen, rSource.left-1,
                             rSource.top-1,
                             rSource.right+1,
                             rSource.bottom+1);
        }
        InvalidateRect (Msg.Receiver, NULL, FALSE);
    }
}
/*
Компонентная функция класса TMagnifyWindow
*/
void TMagnifyWindow::ClientToSource(LPRECT lpIn, LPRECT lpOut)
{
    int xCenter, yCenter;
    int xDelta, yDelta;
    RECT rIn;

    rIn = *lpIn;
    xCenter    = rIn.right / 2;
    yCenter    = rIn.bottom / 2;
    xDelta     = (10 * xCenter) / nStretch;

```



```

yDelta      = (10 * yCenter) / nStretch;
rIn.left     = xCenter - xDelta;
rIn.top      = yCenter - yDelta;
rIn.right    = xCenter + xDelta;
rIn.bottom   = yCenter + yDelta;
*lpOut = rIn;
}

```

```

/*
Компонентная функция класса TNotRect
*/

```

```

TNotRect::TNotRect(int X1, int Y1, int X2, int Y2)

```

```

{
    bVisible = FALSE;
    rCurrent.left    = X1;
    rCurrent.top     = Y1;
    rCurrent.right    = X2;
    rCurrent.bottom   = Y2;
}

```

```

/*
Компонентная функция класса TNotRect
*/

```

```

void TNotRect::Hide(HDC hdc)

```

```

{
    if (hdc == 0 || !bVisible)
        return;

    Invert (hdc);
    bVisible = FALSE;
}

```

```

/*
Компонентная функция класса TNotRect
*/

```

```

void TNotRect::Show (HDC hdc)

```

```

{
    if (hdc == 0 || bVisible)
        return;

    Invert (hdc);
    bVisible = TRUE;
}

```

```

/*
Компонентная функция класса TNotRect
*/

```

```

void TNotRect::Move (HDC hdc, int X1, int Y1, int X2, int Y2)

```

```

{
    if (hdc == 0 || !bVisible)
        return;
}

```



```

Invert (hdc);
rCurrent.left      = X1;
rCurrent.top       = Y1;
rCurrent.right     = X2;
rCurrent.bottom    = Y2;
Invert (hdc);
}

```

/*

Компонентная функция класса TNotRect

*/

```

void TNotRect::Invert(HDC hdc)
{
    int ropOld;

    ropOld = SetROP2 (hdc, R2_NOT);

    MoveTo (hdc, rCurrent.left, rCurrent.top);
    LineTo (hdc, rCurrent.right, rCurrent.top);
    LineTo (hdc, rCurrent.right, rCurrent.bottom);
    LineTo (hdc, rCurrent.left, rCurrent.bottom);
    LineTo (hdc, rCurrent.left, rCurrent.top);

    SetROP2 (hdc, ropOld);
}

```

MAGNIFY.RC

snapshot icon Magnify.ico

hand cursor Magnify.cur

MAGNIFY.DEF

NAME MAGNIFY

EXETYPE WINDOWS

DESCRIPTION 'Stretch parts of the display screen.'

CODE MOVEABLE DISCARDABLE

DATA MOVEABLE MULTIPLE

HEAPSIZE 512

STACKSIZE 5120

Содержание

Предисловие	3
Вводная часть	4
	7
Часть 1. Введение в Windows	7
Глава 1. Введение в Windows	7
История Windows	7
Операционная система Windows NT	11
Проблемы, связанные с программированием под Windows	12
Сообщения и планирование программ	16
Аппаратно-независимая графика	18
Окно	19
Пиктограммы	20
Меню	21
Полосы прокрутки	23
Курсоры	23
Каретка	24
Блоки диалога	24
Элементы управления блоком диалога	25
Часть 2. Минимальная программа для Windows	26
Глава 2. Минимальная программа для Windows	26
Механизм компиляции и компоновки MIN.EXE	31
Утилита MAKE	31
Ключи компилятора	33
Файл ресурсов	34
Компоновщик	35
Компоновщик и файл определения модуля	36
Глава 3. Соглашения о программировании для Windows и OWL	39
Венгерская Запись Имен	39
Соглашения об именах OWL	42
Логические номера	42
Включаемые файлы OWL	43
Включаемый файл Windows	44
Символические константы	44
Определения типов данных	45
Прототипы функций	46
Устаревшая практика: Приведение типов	47
Сообщения	48
Глава 4. Объект "Прикладная программа"	49
Объявление процедуры WinMain	49
Класс TModule	51
Класс TApplication	53
Класс TMinApplication программы MIN	56
Стандартный цикл опроса сообщений	59
Цикл опроса сообщений OWL	61
Глава 5. Классы оконных объектов из библиотеки OWL	63
Класс TWindowsObject	63
Класс TWindow	67
Класс TMinWindow программы MIN	68
Создание окон в MS Windows	68

Создание окна	71
Создание окна и библиотека OWL	73
Объявление оконной процедуры	74
Функции-обработчики сообщений из библиотеки OWL	75
Завершение программы	77
Устанавливаемая по умолчанию обработка сообщений	79
Систематика сообщений	80
Аппаратные сообщения	81
Сообщения об организации окна	83
Сообщения об организации интерфейса пользователя	85
Сообщения о завершении	86
Частные сообщения	86
Уведомление о системном ресурсе	87
Сообщения о совместном использовании данных	88
Внутренние системные сообщения	89
Часть 3. Введение в интерфейс графических устройств (GDI)	90
Глава 6. Обзор интерфейса графических устройств (GDI)	90
Средства GDI	90
Устройства GDI	91
Программный интерфейс	92
Координаты графического вывода	93
Логические объекты графики	94
Контекст устройства	94
Вырезка изображения и менеджер окон	100
Глава 7. Пиксели и маркеры	102
Подпрограмма BeginPaint	105
Подпрограмма GetClientRect	107
Подпрограмма SetPixel	107
Подпрограмма EndPaint	109
Сэндвич Windows	109
Создание маркеров	111
Глава 8. Рисование линий	119
Примитивы для рисования линий	120
Подпрограммы MoveTo и LineTo	120
Атрибуты контекста устройства	126
О перьях	127
Независимость перьев и устройств	127
Создание и использование перьев	128
Режимы графического вывода.	135
Глава 9. Рисование закрашенных фигур	138
Подпрограммы GDI для рисования закрашенных фигур	144
Подпрограммы Polygon и PolyPolygon	145
Подпрограммы Ellipse, Chord, Pie	147
Подпрограммы Rectangle и RoundRect	148
Атрибуты контекста устройства	148
О кистях	150
Создание и использование кистей	150
Глава 10. Вывод текста	159
Примитивы для вывода текста	160
Подпрограмма TextOut	161
Подпрограмма ExtTextOut	162

Подпрограмма TabbedTextOut	169
Подпрограмма DrawText	171
Подпрограмма GrayString	172
Атрибуты контекста устройства для вывода текста	173
Цвет	173
Выравнивание текста	175
Задание интервалов между символами	177
Шрифты	178
Подпрограмма GetTextExtent	179
Подпрограмма GetTextMetrics	180
Создание и использование логических шрифтов	181
Часть 4. Объекты пользовательского интерфейса	189
Глава 11. Команды меню и кодовые комбинации клавиш	189
Стандарты пользовательского интерфейса	189
Вопросы программирования при создании меню	193
Шаблоны для меню	194
Программа-пример Stanmenu	197
Подпрограммы поддержки меню	203
Подключение меню к окну	207
Уничтожение меню	208
Внесение изменений в меню	209
Запросы	213
Подпрограммы захвата	216
Кодовые комбинации клавиш	217
Трансляция кодовых комбинаций клавиш	223
Глава 12. Расширение графических возможностей меню	229
Элементы пользовательской прорисовки меню	229
Сообщение WM_MEASUREITEM	230
Сообщение WM_DRAWITEM	232
Пример программы: Owndraw	234
Битовые образы в элементах меню	244
Создание контрольных маркеров меню владельца	263
Приложение С. Глоссарий	275
Приложение G. Установка Интегрированной Среды Разработчика	293
Приложение Н. Программа MAGNIFY	301

Издательство "ДИАЛЕКТИКА"
предлагает Вашему вниманию:

Буч Г. **Объектно-ориентированное проектирование с примерами применения**, 528 с., твердый переплет. Книга вышла в свет в 1992 г.

ObjectWindows для C++. — В 2-х томах по 280 с., мягкий переплет, слайд. Книга вышла в свет в 1992 г.

Turbo Vision для языка Паскаль. — В 2-х томах по 320 с., мягкий переплет, слайд. Книга вышла в свет в 1993 г.

Справочное руководство по FoxPro 2.0. — В 3-х томах по 620 с., твердый переплет. Книга вышла в свет в 1993 г.

Федоров А., Рогаткин Дм., **Borland Pascal в среде Windows**, 512 с., твердый переплет. Книга вышла в свет в 1993 г.

Resource Workshop. Проектирование ресурсов Windows-приложений, 224 с., мягкий переплет. Книга вышла в свет в 1993 г.

Федоров А., **Borland Pascal: практическое использование Turbo Vision 2.0**, 260 с., мягкий переплет, слайд. Книга вышла в свет в 1993 г.

Справочник по процедурам и функциям Borland Pascal 7.0, 260 с., мягкий переплет. Ориентировочный выход в свет - декабрь 1993 г.

Справочник по функциям Borland C++ 3.1-4.0, 300 с., мягкий переплет. Ориентировочный выход в свет - декабрь 1993 г.

Шлеер С., Меллор С., **"Объектно-ориентированный анализ: моделирование мира в состояниях"**, 300 с., мягкий переплет. Ориентировочный выход в свет - январь 1994 г.

Использование Turbo Assembler при разработке программ, 288 с., мягкий переплет. Ориентировочный выход в свет - январь 1994 г.

Особенности программирования на Borland Pascal, 260 с., мягкий переплет. Ориентировочный выход в свет - январь 1994 г.

Ливингстон Б., **Секреты Windows 3.1**. — В 2-х томах по 350 с., мягкий переплет, слайд, совместное издание "Диалектика" & ICE. Ориентировочный выход в свет - март 1994 г.

Буч Г. **Объектно-ориентированное проектирование с примерами применения**, 528 с., твердый переплет.

Предлагаемая Вашему вниманию книга "Object Oriented Design with Application" by Grady Booch" (в издании на русском языке - "Объектно-ориентированное проектирование с примерами применения") вышла в издательстве The Benjamin/Cummings Publishing, Inc. (США) в 1991 году и является одной из самых продаваемых.

Об авторе. Гради Буч - основатель и директор американской фирмы Object-Oriented Products at Rational, образованной в 1980 году. Был одним из первых в США, кто начал использование объектно-ориентированного метода проектирования с применением различных объектных и объектно-ориентированных языков программирования. Г-н Буч является членом редколлегии американских журналов: Object Magazin, Journal of Object-Oriented Programming, HotLine of Object-Oriented Technology.

О книге. "Объектно-ориентированное проектирование с примерами применения" представляет собой первое полное изложение объектно-ориентированной методологии и ее компонентов. Книга разделена автором на три основные части.

Часть первая - КОНЦЕПЦИИ. В этой части обсуждаются свойства сложных систем, раскрывается понятие декомпозиции, уточняется роль абстракции и иерархии. Конструирование системы определяется как процесс создания модели реального объекта. Автором рассмотрены фундаментальные основы объектного подхода и произведен анализ развития языков программирования. В качестве элементов объектного подхода выделяются процессы объектно-ориентированного анализа (ООА), проектирования (ООД) и программирования (ООР). Перечислены направления, где практические результаты использования этого подхода уже получены. Подчеркивается значение Классов и Объектов в процессе проектирования и даются рекомендации по оценке качества этих абстракций. Приводятся примеры структур классов и объектов, оптимизирующих программные системы. Отдельная глава посвящена выработке правил классификации классов и объектов, рассматриваются три основных типа классификаций:

- классическое распределение по категориям;
- концептуальное группирование;
- теория прототипирования.

Часть вторая - МЕТОДОЛОГИЯ. Здесь речь идет о системе обозначений, позволяющей описать проект, не прибегая к средствам языка реализации проекта. Автором описан собственно процесс объектно-ориентированного проектирования, выделены основные этапы этого процесса, рассмотрены содержание и результаты каждого из них. Выделены практические аспекты конструирования реальных программных систем, названы основные инструменты, облегчающие труд программиста. Даны некоторые советы, как перейти к объектно-ориентированному проектированию.

Часть третья - ПРИМЕРЫ ПРИМЕНЕНИЯ. В этой части приводятся примеры применения на следующих языках программирования: Smalltalk, Object Pascal, C++, Common Lisp Object System, Ada.

Особое достоинство книги - обширная библиография на сорока страницах, в которой представлены публикации по всем аспектам технологии объектно-ориентированного программирования.

Книга рассчитана на профессиональных программистов, руководителей больших программных проектов и студентов, будущая профессиональная деятельность которых связана с разработкой сложных программных систем.

ObjectWindows для C++. — В 2-х томах по 280 с., мягкий переплет, слайд.

Содержит в себе описание и справочный материал по объектно-ориентированной библиотеке OBJECTWINDOWS. ObjectWindows обеспечивает совершенно новый великолепный способ создания прикладных программ для Microsoft Windows.

ObjectWindows упрощает создание программ для Windows благодаря:

- согласующемуся интуитивному и упрощенному интерфейсу с Windows;
- механизму управления окнами и обработки сообщений;

- базовому набору оконных элементов для структурирования
- прикладной программе для Windows.

Вы автоматически получаете эти базовые функциональные возможности, которые позволяют Вам сосредоточить основные усилия на специфических требованиях Вашей прикладной программы.

Детально рассмотрены принципы написания прикладных программ для Windows.

Книга написана в стиле учебного пособия, где раскрывается весь процесс использования и расширения возможностей данной библиотеки. Содержится описание элементов самой библиотеки, включая обзор иерархии классов и объяснения по взаимодействию со средой Windows. Рассматриваются аспекты программирования в Windows, которые ObjectWindows не поддерживает: графика, связь между процессами, использование ресурсов, управление памятью, разделение кодов и данных и управляющие строки в Windows.

Turbo Vision для языка Паскаль. — В 2-х томах по 320 с., мягкий переплет, слайд.

Книга описывает объектно-ориентированную библиотеку - программный продукт нового поколения, в котором впервые на базе процедурного языка реализована концепция событийного программирования. В этой книге наглядно демонстрируется то, как следует проектировать и реализовывать систему взаимодействующих объектов, а также показан механизм записи функционирующей системы объектов на диск. Описываемое инструментальное средство предназначено для проектирования интерактивной интерфейсной оболочки пользователя в текстовом режиме в соответствии со стандартом SAA/CUI.

Книга содержит полную справочную информацию о всех стандартных объектах и методах объектно-ориентированной библиотеки Turbo Vision.

Предназначена для программистов разной квалификации, для всех использующих язык программирования Turbo Pascal.

Справочное руководство по FoxPro 2.0. - В 3-х томах по 620 с., твердый переплет.

В трехтомнике приводится полное описание одной из наиболее мощных реляционных систем управления базами данных для персональных компьютеров - FoxPro 2.0. Содержится большое количество иллюстраций и примеров программ, которые облегчают освоение ее функциональных возможностей.

Федоров А., Рогаткин Дм. ***Borland Pascal в среде Windows***, 512 с., твердый переплет.

Поздравляем Вас с решением заниматься программированием под Windows. Поскольку этот продукт в 1990-1992 гг. наделал много шума, то можно считать, что через несколько лет почти каждая программа будет работать под Windows.

Почти 10 лет назад фирма Borland выбросила на рынок свой легендарный компилятор Turbo Pascal. В то время память компьютеров считалась еще в килобайтах и тактовая частота не превышала 10 MHz. Уже тогда Turbo Pascal обгонял свое время, обладая необычайным комфортом и очень высокой скоростью компиляции, поэтому название Turbo Pascal происходит от слова Turbo, что означает быстрый. Появление на рынке программных средств очередной версии компилятора Borland Pascal говорит о том, что это направление является приоритетным для фирмы Borland.

Книга посвящена основам программирования в среде Microsoft Windows на языке Pascal, но обсуждение многих общих аспектов программирования, особенно интерфейса Windows API, также может быть полезно пользователям компиляторов Borland C++, Turbo C++ for Windows и Microsoft C совместно с Microsoft Windows SDK.

Что Вас ждет в этой книге ?

Авторы детально описывают процесс создания Windows-программы на основе функций Windows API и объектно-ориентированной библиотеки ObjectWindows.

Книга состоит из следующих основных разделов:

- "Компилятор Turbo Pascal 1.5";
- "Программная организация Windows";
- "Введение в элементы интерфейса Windows";
- "Простейшая программа для Windows";
- "Сообщения";
- "Введение в библиотеку ObjectWindows";
- "Окно";
- "Меню";
- "Панель диалога";
- "Элементы управления";
- "Ввод данных. Клавиатура и мышь";
- "Работа с манипулятором "мышь";
- "Введение в GDI";
- "Курсоры, иконки и растровые изображения";
- "Работа с текстом";
- "Использование принтера";
- "Работа с файловой системой";
- "Динамические библиотеки";
- "Управление памятью";
- "Средства обмена данными";
- " Borland Pascal 7.0. Краткий обзор".

В книге в очень компактной и сжатой форме собрана обширная информация по технологии разработки приложений для Windows, что может сделать ее незаменимой при практической разработке программ.

Resource Workshop. Проектирование ресурсов Windows-приложений, 224 с., мягкий переплет.

Данное руководство поясняет, как использовать Пакет разработчика ресурсов для разработки ресурсов Windows. Здесь не объясняется, как писать программы Windows или как писать исходный код программ, необходимый для доступа к ресурсам. Предполагается, что Вы знакомы с основами программирования в Windows или изучите данные вопросы в других книгах.

В первой части данного руководства рассказывается о ресурсах Windows и о том, как Пакет разработчика ресурсов помогает работать с ними в файлах описания проектов.

Глава 1. "Начало работы". Рассказывает о ресурсах Windows, о том, как установить Пакет разработчика ресурсов, начать работу, завершить сеанс работы с ним и как использовать его подсказку.

Глава 2. "Основы работы с Пакетом разработчика ресурсов". Содержит краткий обзор ресурсов и типов файлов, с которыми может работать Пакет разработчика ресурсов, описание видов редакторов, использующихся для их редактирования. Здесь же объясняется, как использовать указанные файлы в процессе построения ресурсов для прикладных программ Windows и какие ресурсы можно в них сохранять. Вводится понятие файла описания проекта, который включает в себя все ресурсы данной программы.

Глава 3. "Работа с проектами, ресурсами и идентификаторами". Проекты рассматриваются более подробно, рассказывается о задании и использовании файлов описания проектов, редактировании и добавлении ресурсов и координации используемых в ресурсах идентификаторов с теми идентификаторами, которые используются в Ваших программах.

В оставшихся главах речь идет о различных редакторах ресурсов, содержащихся в Пакете разработчика ресурсов.

Глава 4. "Создание диалоговых окон". Посвящена описанию редактора диалогов, включая все аспекты создания и модификации диалоговых окон и элементов управления, которые они содержат. Редактор диалогов разрабатывает, модифицирует и проверяет диалоговые окна вне Ваших программ.

Глава 5. "Создание меню". Об использовании редактора меню - визуального средства, позволяющего создавать, модифицировать и проверять спускающиеся и всплывающие меню в правой части экрана.

Глава 6. "Создание акселераторов". Предлагается вниманию редактор, который создает акселераторы - командные комбинации клавиш в прикладной программе Windows. С помощью редактора акселераторов Вы можете назначать клавиши для команд Вашей прикладной программы (оперативные клавиши).

В Главе 7 "Создание строковой таблицы". Объясняется, как использовать строковый редактор, который дает Вам возможность создавать и поддерживать строковую таблицу ресурсов. Данные ресурсы содержат группу строк, представляющих собой используемый прикладной программой текст подсказок и сообщений об ошибках.

Глава 8. "Использование графического редактора". Даются основы работы с графическим редактором, который Пакет разработчика ресурсов запускает при выборе ресурса в виде битового массива (пиктограммы, курсора, битового массива или шрифта). Представлены средства графического редактора, общие для всех графических ресурсов, а со спецификой работы с конкретным графическим ресурсом можно ознакомиться в следующей главе.

В главах 9 - 12 подробно описывается редактирование пиктограмм, курсоров, битовых массивов и шрифтов.

- Пиктограммы (Глава 9. "Создание пиктограмм") - это небольшие квадратные графические образы, используемые для представления программ или минимизированных окон.
- Курсоры (Глава 10. "Создание курсоров") представляют собой графические образы, которые служат для указания позиции стрелки "мыши" на экране.
- Битовые массивы (Глава 11. "Создание битовых массивов") - это различные графические образы, имеющие широкую область применения, включая создание фона, картинок и специальных управляющих элементов.
- Шрифты (Глава 12. "Создание шрифтов") - это наборы графических образов для представления текстовых символов. Шрифты также могут быть наборами битовых массивов, хранящихся в более экономичной форме.

Глава 13. "Создание ресурсов, определенных пользователем". Поясняется, как использовать другие виды ресурсов, которые Вы можете определить. Все ресурсы, описанные в предыдущих главах, являются стандартными - определяемыми и обрабатываемыми Windows.

Если данные ресурсы не отвечают Вашим требованиям, Вы можете использовать ресурсы, определяемые пользователем.

"Технические замечания" Приложения А касаются различных аспектов Пакета разработчика ресурсов, включая совместимость с компилятором ресурсов фирмы Microsoft, использование диалоговых окон в качестве порожденных окон и прочие специальные вопросы. Там также описаны три утилиты: RLINK, BRCC и RCC.

В Приложении В "Borland Windows Custom Controls (BWCC)" описан BWCC, который Вы можете использовать для расширения функциональности и улучшения внешнего вида Ваших диалоговых окон. Даются советы, как можно настроить уже написанные программы под Windows на использование BWCC.

Федоров А. ***Borland Pascal: практическое использование Turbo Vision 2.0***, 260 с., мягкий переплет, слайд.

Данная книга посвящена практическому использованию объектно-ориентированной библиотеки Turbo Vision, разработанной фирмой Borland International Inc. Появившаяся впервые в комплекте с компилятором Turbo Pascal 6.0 эта библиотека завоевала огромную популярность среди отечественных программистов. Последующий перенос Turbo Vision в среду компилятора Turbo C++ сделал ее доступной и для любителей этого языка.

Книга обобщает более чем двухлетний опыт автора по использованию и самой библиотеки, и объектно-ориентированного расширения языка Pascal. Предполагается, что читатели знакомы с основными принципами организации Turbo Vision и име-

ют опыт ее практического использования. Книга должна рассматриваться как приложение к стандартной документации, ни в коей мере не заменяющего ее.

Эта книга отличается от предыдущих, посвященных данной теме, тем, что в ней Вы не найдете ни длинных иерархий объектов, ни подробных описаний отдельных методов или объектов - для этого существует документация или справочная система. Автор сосредоточил свое внимание на практической стороне - Вы обнаружите большое число примеров использования различных объектов, примеры расширения функциональности объектов и создания новых объектов на базе уже существующих. Там, где это необходимо, приводятся некоторые теоретические сведения, но в основном это набор практических советов.

Появление новой версии Turbo Vision в составе компилятора Turbo Pascal/Borland Pascal 7.0 еще раз доказало, что заложенные в ней концепции долговечны. Пока будет использоваться среда DOS, Turbo Vision будет жить. Существующие реализации Turbo Vision в графическом режиме еще больше расширяют функциональность этой библиотеки, приближая ее к Microsoft Windows.

Книга построена по принципу: от простого - к сложному. Вначале рассматриваются основные концепции, необходимые для понимания функционирования библиотеки в целом. Отдельные главы посвящены группам однородных объектов: рассматриваются способы их применения и примеры расширения. Каждый пример является законченной программой - такой подход кажется автору более правильным, чем, например, построение одной большой программы из частей: в результате Вы теряете мысль, а листинг, растянутый на десятки страниц, становится просто бесполезным. В приложение вынесены те материалы, которые могут пригодиться пользователям Turbo Vision, но не оформлены должным образом в документации.

Справочник по процедурам и функциям Borland Pascal 7.0, 260 с., мягкий переплет.

Книга содержит описание включенных в систему программирования Borland Pascal with Objects 7.0 всех процедур, функций, типов данных, переменных и констант для их использования в реальном, защищенном и Windows-режиме (без Turbo Vision и ObjectWindows).

Справочник по функциям Borland C++ 3.1, 300 с., мягкий переплет.

Справочник содержит описание всех библиотечных функций, глобальных переменных и типов Borland C++ 3.1. В книге также содержится перечень функций, сгруппированных по модулям и назначениям.

Шлеер С., Меллор С. ***Объектно-ориентированный анализ: моделирование мира в состояниях***, 250 стр., мягкая обложка.

Объектно-ориентированная методология (ООМ) и ее компоненты: объектно-ориентированный анализ (ООА), объектно-ориентированное проектирование (ООД) и объектно-ориентированное программирование (ООР) - стали в настоящее время одной из самых популярных тем в области информатики. Известно, что благодаря объектно-ориентированной методологии решение проблем разработки сложного программного обеспечения существенно упрощается. Однако вопросам реализации или собственно языкам программирования придается сегодня преувеличенное значение в ущерб вопросам высокоуровневого проектирования программных систем.

Предлагаемая Вашему вниманию книга известных американских специалистов С.Шлеер и С.Меллора посвящена изложению самых первых этапов процесса разработки сложных (программных, технических или других) систем. В ней детально описывается один из наиболее нетривиальных элементов объектного подхода - процесс объектно-ориентированного анализа. Методология ООА разработана в книге как метод для отождествления важных сущностей в задачах реального мира, для объяснения и понимания того, как они взаимодействуют между собой. Изложенное представление ООА целенаправленно адресовано программистам-профессионалам и всем, кто занимается программированием. Данный метод анализа изложен в виде примеров и руководящих принципов, которые могут быть усвоены довольно быстро.

После деления системы программного обеспечения на ряд четко определенных предметных областей каждая из них анализируется независимо от остальных в три этапа. В процессе информационного моделирования выделяются концептуальные сущности (объекты), которые составляют подсистему для анализа. С помощью моделей состояний исследуется поведение идентифицированных объектов и связей между ними во времени. Каждое действие, происходящее в системе, рассматривается на этапе моделирования процессов. Созданные модели в дальнейшем могут быть непосредственно преобразованы в объектно-ориентированный проект с помощью рекурсивного проектирования (RD), описанного в книге. Авторы также приводят не зависимую от языка графическую нотацию для описания проектирования программы, библиотеки или среды. Завершает книгу краткий англо-русский словарь терминов, наиболее часто употребляемых при объектно-ориентированном подходе.

Использование Turbo Assembler при разработке программ, 288 с., мягкий переплет.

В книге описывается одна из наиболее распространенных систем программирования на языке Ассемблер для персональных компьютеров - Turbo Assembler 3.x (TASM). Приводится большое число примеров программ и обширная справочная информация по системе TASM. Книга состоит из восьми частей. Материал излагается в порядке возрастания сложности, поэтому те, кто только начинает изучать Ассемблер, могут переходить сразу к изучению нужных им тем.

Часть первая содержит первоначальные сведения об Ассемблере, описание архитектуры процессора 8086, основы организации загрузки и выполнения программ в DOS, а также подробное описание инструкций процессора 8086.

Часть вторая посвящена применению Ассемблера для разработки программ в реальном режиме.

Часть третья описывает архитектуру процессора 80386, инструкции защищенного режима и возможности Турбо Ассемблера по написанию программ, использующих защищенный режим.

Часть четвертая подробно излагает основные понятия технологии объектно-ориентированного программирования, содержит описание директив Турбо Ассемблера и примеры программ, реализующих эту технологию.

Часть пятая предназначена для программистов, работающих на языках высокого уровня, и описывает интерфейсы между Турбо Ассемблером и компиляторами с языков C++, Pascal, Basic и Prolog, разработанными фирмой Borland Int.

Часть шестая излагает основные концепции программирования в среде Microsoft Windows, содержит примеры программ для этой среды.

Часть седьмая посвящена описанию архитектуры и инструкций математических сопроцессоров x87, содержит примеры программ, использующих сопроцессор.

Часть восьмая носит справочный характер, содержит описание основных прерываний DOS и Windows, краткий перечень всех процессоров x86 и сопроцессоров x86, описание форматов исполняемых файлов для DOS и Windows, перечень сообщений, генерируемых Турбо Ассемблером в случае обнаружения ошибки в тексте программы, и другие сведения по практическому применению Турбо Ассемблера для разработки программ.

Во втором томе книги П.Нортон и П.Йао "**Программирование на Borland C++ в среде Windows**" Вы найдете аннотации на другие книги, намеченные к выходу в свет издательством "Диалектика". Кроме того, Вы сможете узнать об условиях подписки на журнал "SoftReview / Компьютерное обозрение", приобретения литературы по компьютерной тематике.

SOFTREVIEW

компьютерное
обозрение

Международный
журнал
для программистов

Методология объектного подхода
Языки программирования
Библиотеки
Инструментальные средства
Спектакль кода
Библия программиста
Есть такое мнение
Новости, события, интервью, реклама

Издатель журнала:
научно-исследовательская
проектная фирма
"Диалектика"



Украина, 252124, Киев-124, а/я 506



(044) 266-40-74

